T.C.
İSTANBUL BEYKENT ÜNİVERSİTESİ
LİSANSÜSTÜ EĞİTİM ENSTİTÜSÜ

BİLGİSAYAR MÜHENDİSLİĞİ ANABİLİM DALI
BİLGİSAYAR MÜHENDİSLİĞİ BİLİM DALI

MİKROSERVİS TABANLI E-TİCARET UYGULAMALARINDA SİPARİŞ ODAKLI ENDPOİNT YÖNETİMİ

Yüksek Lisans Tezi

Tezi Hazırlayan
Hilal ŞEN

İstanbul, 2024

T.C.
İSTANBUL BEYKENT ÜNİVERSİTESİ
LİSANSÜSTÜ EĞİTİM ENSTİTÜSÜ

BİLGİSAYAR MÜHENDİSLİĞİ ANABİLİM DALI
BİLGİSAYAR MÜHENDİSLİĞİ BİLİM DALI

MİKROSERVİS TABANLI E-TİCARET UYGULAMALARINDA SİPARİŞ ODAKLI ENDPOİNT YÖNETİMİ

Yüksek Lisans Tezi

Tezi Hazırlayan
Hilal ŞEN

Öğrenci No
2120003035

ORCID ID
0009-0008-0845-6981

Danışman
Dr. Öğr. Üyesi Talat FİRLAR

İstanbul, 2024

YEMİN METNİ

Yüksek Lisans Tezi olarak sunduğum "**Mikroservis Tabanlı E-Ticaret Uygulamalarında Sipariş Odaklı Endpoint Yönetimi**" başlıklı bu çalışmanın, bilimsel ahlak ve geleneklere uygun şekilde tarafımdan yazıldığını, bu tezdeki bütün bilgileri akademik ve etik kurallar içinde elde ettiğimi, yararlandığım eserlerin tamamının kaynaklarda gösterildiğini ve çalışmamın içinde kullanıldıkları her yerde bunlara atıf yapıldığını, patent ve telif haklarını ihlal edici bir davranışımın olmadığını belirtir ve bunu onurumla doğrularım. 15/04/2024

Hilal ŞEN

T.C.
İSTANBUL BEYKENT ÜNİVERSİTESİ
LİSANSÜSTÜ EĞİTİM ENSTİTÜSÜ MÜDÜRLÜĞÜ
TEZLİ YÜKSEK LİSANS SINAV TUTANAĞI

11/06/2024

Enstitümüz ***Bilgisayar Mühendisliği*** Anabilim Dalı ***Bilgisayar Mühendisliği*** Programı yüksek lisans öğrencilerinden ***2120003035*** numaralı ***Hilal ŞEN***'in *"İstanbul Beykent Üniversitesi Lisansüstü Eğitim – Öğretim Yönetmeliği"*nin ilgili maddesine göre hazırlayarak, Enstitümüze teslim ettiği ***"Mikroservis Tabanlı E- Ticaret Uygulamalarında Sipariş Odaklı Endpoint Yönetimi"*** konulu tezini, Yönetim Kurulumuzun 28/05/2024 tarih ve 2024/24 sayılı toplantısında seçilen ve On-Line toplanan biz jüri üyeleri huzurunda, İstanbul Beykent Üniversitesi Lisansüstü Eğitim ve Öğretim Yönetmeliğinin 29. maddesinin 3. fıkrası gereğince 45 dakika süre ile Zoom programı aracılığıyla on-line olarak aday tarafından savunulmuş ve sonuçta adayın tezi hakkında ***"OYBİRLİĞİ"*** ile ***"KABUL"*** kararı verilmiştir.

İşbu tutanak, 2 nüsha olarak hazırlanmış ve Enstitü Müdürlüğü'ne sunulmak üzere tarafımızdan düzenlenmiştir.

DANIŞMAN
Dr. Öğr. Üyesi Ta*** Fİ***
(İstanbul Beykent Üniversitesi)

ÜYE
Dr. Öğr. Üyesi Ke*** Gö*** NA***
(İstanbul Beykent Üniversitesi)

ÜYE
Doç Dr. Em*** SE***
(İstinye Üniversitesi)

Adı ve Soyadı : Hilal ŞEN
Danışman : Dr. Öğr. Üyesi Talat FİRLAR
Derecesi ve Tarihi : Yüksek Lisans (Tezli), 2024
Sanatta Yeterlilik Alanı : Bilgisayar Mühendisliği
Anahtar Kelimeler : Mikroservis Mimari, E-ticaret, Hata Mekanizmaları, Sipariş Hataları,

ÖZ

MİKROSERVİS TABANLI E-TİCARET UYGULAMALARINDA SİPARİŞ ODAKLI ENDPOİNT YÖNETİMİ

Günümüz teknolojisindeki hızlı gelişmeler, bireylerin gereksinimlerini ciddi oranda dönüştürmektedir. Teknolojinin bireyler üzerindeki etkisi baz alınarak dünyada her gün yeni gelişmeler yaşanmaktadır. Modern yazılım projelerinde ihtiyaçlara sürekli yenilerinin eklenmesi, ölçeklenebilirlik, hata izolasyonu, teknoloji çeşitliliği, esneklik gibi konularda sorunlara yol açmaktadır. Monolitik mimari'den günden güne mikroservis mimariye geçiş kaçınılmaz hale gelmektedir. Projelerin büyümesi ile mikroservis mimarilerin kullanım avantajlarından daha fazla yararlanılır hale gelinmiştir. Bu nedenle, monolitik mimari yerine mikroservis mimarisine geçiş, projelerin geleceğe yönelik hale getirilmesine imkan sunmaktadır.

Covid-19 salgını birçok sektörde olduğu gibi; örneğin; sağlık ve tıp malzemeleri, lojistik ve kargo, telekomünikasyon vs. başta olmak üzere e-ticaret sektöründe de büyük bir büyüme yaşandı. Tüketicilerin kolayca ihtiyaçları olan ürünlere ulaşımını kolaylaştırmak amacıyla, hızlı kargo, temassız teslimat ve temassız ödeme gibi seçenekler hayatımıza dahil oldu. Bu kapsamda tüketicilerin siparişlerinin hataya yer vermeden işlenilmesi ve teslimatının sağlanması büyük önem arz etmektedir.

Bu çalışmadaki amaç, e-ticaret uygulamalarında sipariş adımında meydana gelen bir takım hataların önüne geçilerek, maliyet, sistem karmaşıklığı, performans kaybı,

veriler arası tutarlılık gibi konuların iyileştirilmesi sağlanıp tüketicilerin ihtiyaçlarına daha iyi yanıt vermek, daha optimize edilmiş bir deneyim sunulması amaçlanmıştır.

Çalışma kapsamında, mikroservis mimari ile oluşturulmuş e-ticaret uygulamasında yapılan bir istek anında servislerden herhangi birinde alınacak olan hata bütün isteği başarısız kılıyor olup tüketiciler tarafından bakıldığında kabul edilebilir bir sonuç değildir. Oluşan problem üzerinde hata mekanizmalarının uygulanabilirliği, uygulamaların nasıl işleneceği, mekanizmaları kullanmanın kazanımları, kullanılması ve kullanılmaması durumlarında yaşanacak etkenlerin analizlerine yer verilmiştir.

Name and Surname : Hilal ŞEN
Supervisor : Asst. Prof. Dr. Talat FİRLAR
Degree and Date : Master's (Thesis), 2024
Major : Computer Engineering
Keywords : Microservices Architecture, E-commerce, Error Mechanisms,Order Errors

ABSTRACT

ORDER-ORIENTED ENDPOINT MANAGEMENT ON MICROSERVICE-BASED E-COMMERCE APPLICATIONS

The rapid advancements in today's technology significantly transform the needs of individuals. Considering the impact of technology on individuals, new developments are occurring every day worldwide. In modern software projects, continuous additions to meet requirements, scalability, error isolation, technology diversity, and flexibility pose challenges. The transition from a monolithic architecture to microservices architecture is becoming inevitable day by day. As projects grow, the advantages of using microservices architecture are increasingly realized. Therefore, transitioning from a monolithic architecture to microservices architecture provides an opportunity to make projects more future-oriented.

The Covid-19 pandemic has led to significant growth in various sectors, such as healthcare and medical supplies, logistics and shipping, telecommunications, and especially in the e-commerce sector. To facilitate consumers 'access to the products they need, options like fast delivery, contactless delivery, and contactless payment have become a part of our lives. Within this context, processing and delivering consumers 'orders without errors is of great importance.

The aim of this study is to prevent some errors occurring in the order step in e-commerce applications, improve issues such as cost, system complexity, performance

loss, data consistency, and provide better responses to consumers ‘needs, offering a more optimized experience.

Within the scope of the study, when a request is made in an e-commerce application created with microservices architecture, if an error occurs in any of the services at the moment of the request, it renders the entire request unsuccessful, which is not an acceptable outcome from the consumers ‘perspective. The feasibility of error mechanisms for the problem that arises, how applications will be processed, the benefits of using these mechanisms, and an analysis of the factors to be experienced when using or not using these mechanisms are included.

İÇİNDEKİLER

Sayfa No.

3.UYGULAMA

TABLOLAR LİSTESİ

Sayfa No.

ŞEKİLLER LİSTESİ

Sayfa No.

KISALTMALAR

ACID : Atomicity, Consistency, Isolation, Durability(Bütünlük, Tutarlılık, Bağımsızlık, Dayanıklılık)

AJAX : Asynchronous JavaScript and XML (Asenkron JavaScript ve XML)

API : Application Programming Interface (Uygulama Programlama Arayüzü)

AWS : Amazon Web Services (Amazon Web Hizmetleri)

CRUD : Create, Read, Update, Delete (Oluştur, Oku, Güncelle,Sil)

DevOps : Development and Operations (Geliştirme ve İşletme)

ESB : Enterprise Service Bus (Kurumsal Veri Yolu)

gRPC : Remote Procedure Call (Uzak Prosedür Çağrısı)

HTML : Hypertext Markup Language (Hipermetin İşaretleme Dili)

HTTP : Hypertext Transfer Protocol (Hipermetin Transfer Protokolü)

HTTPS : Hypertext Transfer Protocol Secure (Güvenli Hipermetin Transfer Protokolü)

IT : Information Technology (Bilgi Teknolojileri)

OOP : Object Oriented Programming (Nesne Yönelimli Programlama)

JSON : JavaScript Object Notation (JavaScript Nesne Notasyonu)

PHP : Hypertext Preprocessor (Hipermetin Önişlemcisi)

POS : Point of Sale (Ödeme Noktası)

RAM : Random Access Memory (Rastgele Erişimli Bellek)

REST :Representational State Transfer (Temsili Durum Aktarımı)

SOA :Service Oriented Architecture (Hizmet Odaklı Mimari)

SSL :Secure Sockets Layer (Güvenli Yuva Katmanı)

TCP :Transmission Control Protocol (Aktarım Kontrol Protokolü)

URI : Uniform Resource Identifier (Tekdüzen Kaynak İşaretleyicisi)

URL :Uniform Resource Locator (Tekdüzen Kaynak Bulucu)

Web :World Wide Web (Dünya Çapında Ağ)

WebAPI :Web Application Programming Interface (Web Uygulama Programlama Arayüzü)

XML :Extensible Markup Language (Genişletilebilir İşaretleme Dili)

SÖZLÜK

Application: Belirli bir işlevi gerçekleştirmek üzere oluşturulmuş yazılım programıdır.

Asp.Net Web Api: Http protokolü ile web servislerin oluşturulması, dağıtılması, kullanılmasını sağlamaktadır.

EndPoint: URL veya IP adresi şeklinde ifade edilmektedir. Servisin sağlamış olduğu API'ye erişilmesi için kullanılan adrestir.

Framework: Bir programlama dili ya da kullanılan teknolojinin standart kütüphaneler, modüller ve bileşenlerini içermektedir. Geliştiricilerin belirli problemleri çözmesi için gereken araçlara sahiptir.

GİRİŞ

Türkiye'de teknolojinin gelişmesiyle birlikte programlamaya olan ilgi ve talep büyük ölçüde arttı. Geçmiş yıllarda monolitik mimari olarak adlandırdığımız, programın bütün işlemlerini büyük bir kod tabanında ele alıp tek bir sunucu üzerinden ayağa kaldırdığımız monolitik mimari yaygın olarak kullanılıyordu. Monolitik mimari yaklaşımı, küçük çaplı projelerde karmaşıklığın düşük olması, kolay yönetilmesi, anlaşılabilirliği dolayısıyla başlıca tercih edilme sebebidir. Projelerin daha büyük kitlelere hitap etmeye başlaması ile beraber karmaşıklığın artması ve projenin tek sunucuda olmasından kaynaklanan çeşitli sorunlar ortaya çıkmaya başladı. Bu nedenle, karmaşıklığın yerini modüler ve yönetimi kolay bir yaklaşım olan mikroservis mimarisi aldı.

Mikroservis mimari, büyük monolitik mimariler yerine daha esnek projeler geliştirilmesine olanak sağlar. ‘’Mikro" kelimesi "ufak" ve "küçük" anlamına gelen yunanca kökenli bir terimdir. Teknoloji alanında da küçüklüğü ifade etmek için kullanılır. "Mikroskop" kelimesi küçük varlıkları daha büyük ve rahat görebilmemize imkan sağlamak anlamına gelir. Büyük ve karmaşık uygulamaları parçala, böl, yönet sistemi ile modüler olarak kodlanmasına imkan sağlar. Her bir mikroservisin kendine ait işlevi ve bu işlevin bağlı olduğu veritabanı bulunmaktadır. Ayrıca her mikroservis kendisi için en performanslı olan programlama dilini ve veritabanını kullanabilir. Her mikroservis kendi işlevini gerçekleştireceği için ayrı ayrı ölçeklendirilebilir. Büyük ölçekli projelerde günümüzde tercih sebebi olarak kullanılmaktadır. Küçük parçalar halinde yönetilmeye en uygun projelerin başında e-ticaret projeleri yer almaktadır.

E-ticaret, internet üzerinden ürün ya da hizmet bedelinin ödemesi yapılarak satın alınması modelini ifade eder. Tipik bir e-ticaret projesinde temel olarak bulunması gereken ürün yönetimi mikroservisi, kullanıcı kimlik doğrulama mikroservisi, fotoğraf yönetimi mikroservisi, sepet mikroservisi, indirim mikroservisi, sipariş yönetimi mikroservisi ve ödeme mikroservisi temel parçalar olarak bulunmaktadır.

E-ticaret projelerinde mikroservislerin yanlış kullanımı ve hatalı uygulanması, firmalara maddi kayıplar, tüketici kayıpları, zaman kayıpları gibi pek çok konuda zarar

vermektedir. Mikroservis mimarisinde hata yönetimi yapılırken hatalar kayıt altına alınmaktadır. Hatalar önceliklendirmesine göre sınıflandırılır ardından ilgili yazılım ekiplerine haber verilir. İş sürekliliğinin sağlanabilmesi için kurtarma stratejileri uygulanır. Örneğin, bir indirim haftasında yoğunluktan dolayı sunucuların kaldıramadığı durumlarda, yük doğrultusunda yedek bir mikroservis hızlı bir şekilde devreye alınabilir. Tüketicilere kesintisiz hizmet sunmak sürdürülebilir e-ticaret projelerinin temel yapı taşıdır.

Günümüzde tüketicilerin alışveriş yaptığı e-ticaret uygulamalarında sipariş işleme süreçleri son derece önemlidir. Ödeme alınması, siparişin işlenmesi, müşteriye ulaştırılması gibi süreçler kritik rol oynamaktadır. Mobil veya Dünya çapında ağ (Web) uygulaması üzerinden satın alınan ürünün sepete eklenmesinin ardından ürün satın alındığında sipariş oluşturma işlemi ilgili mikroservise gönderilir ve hata alınmadığı durumda sipariş oluşturma işlemi sağlanmış olur. Kullanıcının giriş yapmış olduğu verilerin eksik olması, kimlik doğrulamasının başarısız olması veya oturum açılmamış olması, yetki hataları, sipariş limiti, ürün stoğu gibi temel sorunlar siparişin oluşturulması sürecinde karşılaşılabilecek temel hatalardır.

Artan tüketim alışkanlıklarıyla birlikte yaşanan yoğunluk doğrultusunda alınan hataları giderebilmek için farklı çözüm yolları aranmaya başlanmıştır. Microsoft tarafından açık kaynaklı olarak geliştirilen .Net Core, uygulama geliştirmek için kullanılan bir kütüphanedir. Oldukça geniş bir yazılım geliştirici kitlesine sahiptir ve eklentiler, araçlar, kütüphaneler ile sürekli sürümleri güncellenmektedir. .Net Core kütüphanesini kullanarak Uygulama Programala Arayüzü (API) üzerinden e-ticaret firmalarının sipariş yönetimi modülü web uygulamaları, mobil uygulamaları ve tabletler üzerinden erişilebilir hale getirilmektedir.

.Net Core gibi uygulama geliştirilen ortamlar için kütüphaneler bulunur. Kütüphaneler, programlama yapılırken sıklıkla gerekli olan görevleri tekrar tekrar yazmak yerine geliştirme sürecini hızlandırmak için hazır çözümler sunmaktadır. Bu kütüphanelerden biri olan Polly, .Net platformunda hata durumlarını ele almak için oluşturulmuştur. Geliştirilen projelerde uygulamaların güvenilir olması, hata toleransının

yükseltilmesi ve yazılımlar ile beraber sürekli güncel tutulması için bu tür kütüphaneler kullanılır. Hata izleme, hata durumunda uygulamanın kurtarma stratejilerini uygulaması, hata mesajlarının düzenlenmesi, hata anında yedek bir kaynağa yönlendirme yapılması, yeniden isteği tekrarlayabilmesi, tüketiciler açısından büyük bir önem arz etmektedir. Tüketicilerin karşılaştığı işlem hataları olumsuz etkilere sebep olmaktadır. Tüketicilerin memnuniyetsizliği, firmalar açısından uzun vadeli olarak tüketici kaybına sebep olmaktadır.

Mikroservislerin her biri kendi içerisinde bağımsız çalışabilmesiyle beraber kendi hata mekanizmalarına sahip olmaları gerekmektedir. Bu kapsamda, Polly kütüphanesi ile mikroservisler arası iletişimde oluşan hatalar izlenmekte olup kütüphanenin sunmuş olduğu hizmetleri kullanarak mikroservis tabanlı e-ticaret projelerinin daha tüketici dostu olmasına katkı sağlanmaktadır.

Bu tez, mikroservis tabanlı e-ticaret uygulamalarında sipariş yönetim süreçlerinin daha verimli hale getirilmesi ve hataların minimum seviyeye indirilmesi üzerine yapılmış bir çalışmadır. Bu tez kapsamında yapılan çalışma sonucu mikroservis mimari yaklaşımını benimseyerek e-ticaret projeleri geliştirilen firmaların hata yönetimi için oluşturulmuş olan kütüphane kullanımıyla elde edebilecekleri kazanımlar ele alınmıştır.

Mikroservis mimarisi ile hata yönetiminde kullanılan Polly kütüphanesinin e-ticaret uygulamalarında sipariş modülünde kullanılması ve kullanılmaması durumu hakkında kıyaslama yapılmış olup, Polly kütüphanesinin kullanılabildiği yöntemler ve bu yöntemlerin sağlamış olduğu kazanımlar hakkında bilgi verilmiştir. E-ticaret uygulamalarının Polly kütüphanesi ile birlikte kullanılmasının sağladığı faydalardan da bahsedilmiştir. Projelerde hata yönetimi için Polly kütüphanesinin uygulanmasının avantaj ve dezavantajları ele alınmıştır. Ek olarak, mikroservis tabanlı e-ticaret uygulamalarında sipariş yönetiminin yeni teknolojiler ve gelişen yeni yöntemler ile daha verimli hale getirilmesi konusunda öngörülere de yer verilmiştir.

Bu tezi farklı kılan özelliklerden biri, mikroservis mimarileri konu alan tezlerin dışında e-ticaret uygulamalarında kullanılmasının incelenmesidir. Ayrıca, hata yönetimi konusu derinlemesine incelenip, modüler olarak yazılan mikroservislerin sipariş

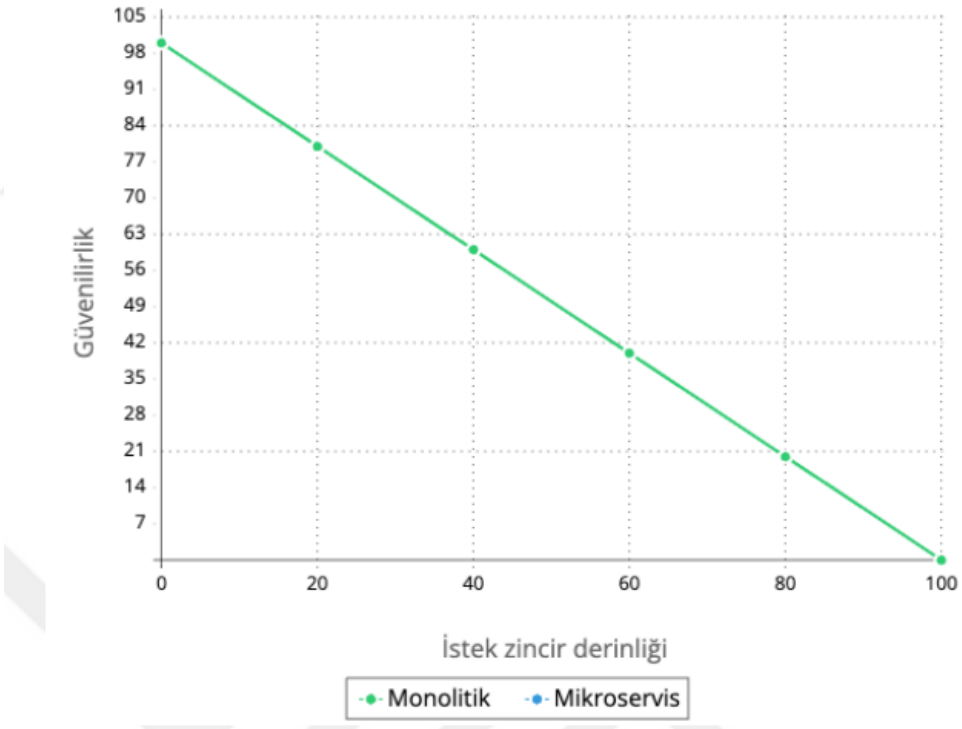
modülünde hata yönetiminin kullanılmasının detaylı analizi ve karşılaştırmaların yapılmış olmasıdır. Ayrıca, hata yönetim kütüphanesi Polly'nin getirdiği çözümlerin kullanılması ile elde edilen kazanımlara da odaklanmaktadır. Böylelikle, günümüzde kullanımı giderek yaygınlaşan e-ticaret uygulamalarının daha kullanılabilir sistemler olarak geliştirilebileceği elde edilmektedir.

Dört bölümden oluşan tezin birinci bölümünde, mikroservis mimari içerisinde e-ticaret uygulamaları ile ilgili benzer çalışmalar yapan kişilerin tezleri incelenmiştir. Tezin ikinci bölümünde, yazılım mimarileri, mikroservis mimari hakkında genel bilgiler, avantajlar ve dezavantajları ele alınmıştır. Ayrıca, e-ticaretin tanımı, kapsamı ve türleri hakkında da bilgi verilmektedir. Üçüncü bölümde, e-ticaret uygulamalarındaki sipariş yönetim süreci ve bu süreçte meydana gelen hatalar ele alınmıştır. Ayrıca, hata yönetiminde kapsamında kullanılan Hipermetin Transfer Protokolü (HTTP) durum kodlarına da değinilmiştir. Çalışma içerisinde kullanılan Polly kütüphanesi hakkında bilgilere sunulmuştur. Dördüncü bölümde ise, e-ticaret uygulamasının sipariş yönetimi modülünde Polly kütüphanesi içerisinde yer alan senaryoların uygulanması veya uygulanmaması durumu ele alınmıştır. Ayrıca, uygulandığı durumda sağladığı faydalar da analiz edilmektedir.

1. LİTERATÜRDE YER ALAN BENZER ÇALIŞMALAR

Dervişi (2022) tarafından yapılan çalışmada, monolitik mimariye sahip olan servislerin mikroservis mimarisine geçiş sürecinde meydana gelen zorluklara odaklanılmış ve bu zorluklara çözüm önerileri ele alınmaktadır. Ayrıca, mikroservis mimarisine aktarılacak verilerle ilgili yapılan araştırmalarda bulunan eksiklerde detaylı bir şekilde ele alınmıştır.

Bankacılık uygulamalarında monolitik mimari ve mikroservis mimari kullanımının getirdiği fayda ve zararlarının ölçeklenebilirlik, gecikme, karmaşıklık, güvenilirlik, kaynak kullanımı ve iletişim konularında değerlendiren bir istek yapılmış ve bu bağlamda elde edilen sonuçlar grafikler halinde sunulmuştur.



Şekil 1. Monolitik Ve Mikroserviste İstek Zincir Derinliğine Karşı Güvenilirlik

Kaynak: Dervişi, F. (2022). *Açık bankacılık sistemlerinde monolitik mimariden mikroservis mimariye geçiş* [Yüksek lisans tezi]. Trakya Üniversitesi.

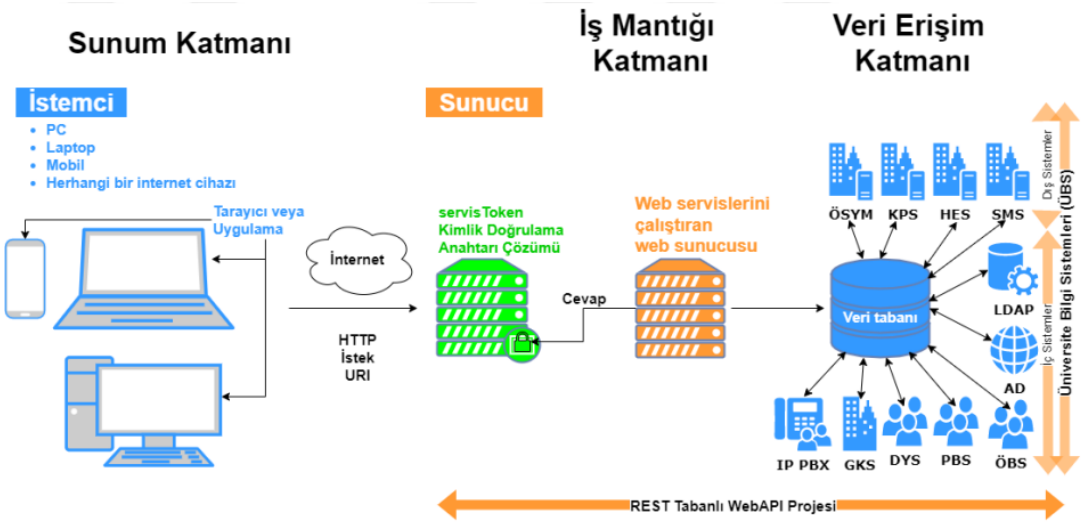
Proje kapsamında, monolitik mimariden mikroservis mimarisine geçiş sürecinde yapılan değişiklikler, toplam sürede azalmaya neden olmuştur. Karmaşıklık açısından, monolitik mimarinin mikroservis mimarisi kadar karışık olmadığı sonucuna varılmıştır. Güvenlik konusunda, ağda yaşanabilecek hatalara karşı mikroservis mimarisinde önlem alınması gerektiğinden bahsedilmiştir. Rastgele Erişimli Bellek (RAM) kullanımı açısından, mikroservis mimarisi daha az RAM tüketmektedir. Ölçeklendirme bakımından, mikroservis mimarisi daha detaylı olarak sonuçlar verdiği için monolitik mimariye göre kullanımı daha avantajlıdır. İletişim açısından parçala, böl ve yönet yaklaşımının ekipler açısından mikroservis mimarisinin monolitik mimariye üstün geldiği konular arasında yer aldığı ifade edilmiştir.

Sunucusuz dağıtım modeli olarak projede seçilmiş olan Amazon Web Hizmetleri (AWS), kullanıcı kaynak esnekliğine sahip olmayı amaçlamaktadır. Temel bankacılık uygulamalarında bulunan kullanıcılar, ödemeler, hesaplar, döviz ve sanal ödeme noktası (POS) apilerinin kullanımı monolitik mimariden mikroservis mimariye taşınmıştır. Bu taşıma sürecinin nasıl daha kolay ve maliyeti düşük tutulabilmesi konusu da incelenmektedir. Firmaların bu geçiş sırasında kalifiye kişilere ihtiyaçları olduğu ve ekip içinde sürecin monolitik mimari kadar kolay olmadığına değinilmiştir.

Bankacılık uygulamalarında yaşanacak hataların güvenlik açığı gibi birçok probleme neden olması olası bir durumdur. Sistemin kolay bakımı ve test edilmesi, monolitik mimariye göre çok daha kolaydır. Yeni istekler geldikçe sürümlerin dağıtılmasına kolaylık tanınmaktadır. Mikroservis mimari kullanımı ile hata ve arızaların minimum seviyeye indirgenmesi ve ekranda gösterilen hataların sürekli izlenebilmesinin daha mümkün olduğundan bahsedilmektedir.

Bu çalışma kapsamında, monolitik mimari ve mikroservis mimari kullanımının kazanımlarından bahsedilmektedir. Bir bankacılık sisteminin monolitik mimariden mikroservis mimarisine geçiş sürecinde dikkat edilmesi gereken hususlar, hata toleransı, güvenlik problemi, servislerin bağımsız çalışabilirliği gibi tercih sebepleri hakkında kıyaslamalara yer verilmiştir.

Altınkaya (2022) tarafından yapılan çalışmada, Üniversite Bilgi Sistemleri için Restful servisler kullanılarak mevcutta kullanılan sisteme erişimi hızlandırmak amacıyla detaylı bir analiz çalışmasıyla geliştirmeler yapılmıştır. Projede, Hizmet Odaklı Mimari (SOA) ve Temsili Durum Aktarımı (REST) mimarisi ayrıntılı şekilde incelenmiştir. SOA mimarisinin kullanım alanları ile REST mimarisininin kullanım alanları ve tercih edilme nedenleri ele alınmıştır. Restful servislerin avantaj ve dezavantajları ile beraber Genişletilebilir İşaretleme Dili (XML), JavaScript Nesne Notasyonu (JSON) vb. standartlar ile kullanılması üzerinde durulmuştur. Üniversite Bilgi Sistemleri, sadece kendi içerisindeki veritabanları ile iletişim halinde değildir. Örneğin, iç sistem servisi olarak Personel Bilgi Sistemi ve dış sistem servisi olarak Ölçme, Seçme ve Yerleştirme Merkezi gibi birçok servis sistem ile iletişim halindedir. Platfom bağımsız şekilde erişim sağlanmaktadır ve kullanıcıların kolay kullanımı, güvenliği gibi konular önem arz etmektedir. Proje katmanlara ayrılarak ilerlenmiştir. Proje sunum katmanı, iş mantığı katmanı, veri erişim katmanı olarak üç katmandan oluşmaktadır.



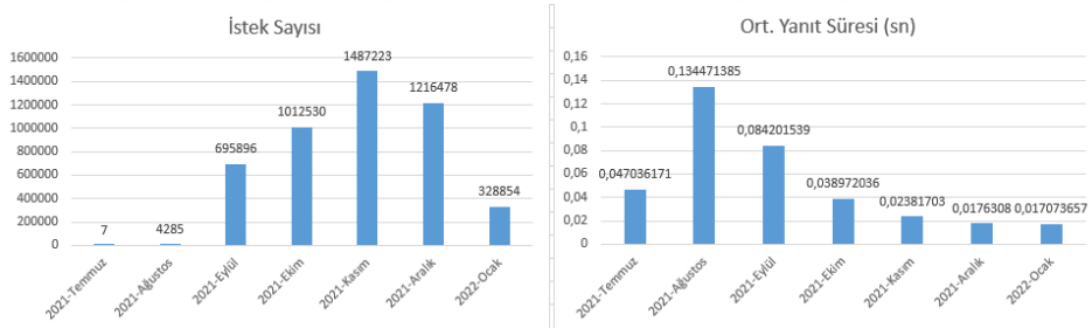
Şekil 2. REST tabanlı WebAPI Projesi Genel Akış Diyagramı

Kaynak: Altınkaya, C. (2022). *Üniversite bilgi sistemleri için REST tabanlı bir web servis platformunun tasarımı ve geliştirilmesi* [Yüksek lisans tezi]. Atatürk Üniversitesi.

Bu proje, MUKimlik projesi örnek alınarak hazırlanmış projedir. MUKimlik projesi, üniversitelerin yeni personelleri ve öğrencilerinin hesap oluşturma işlemlerini dijital sistemler üzerinden gerçekleştirmelerini sağlayan bir sistemdir. Kullanıcılar, sisteme girişteki kullanıcı adı ve parolası ile bu işlemleri tamamlayabilirler.

Proje içerisinde kullanılan kütüphaneler ve sınıflara detaylı bir şekilde yer verilmiştir. Örneğin, proje içerisinde kullanılan HizliSMSGonder, ADHesapOlustur gibi kullanılan tüm metotların parametreleri, dönüş tipleri ve ilgili URL'ler projede belirtilmiştir.

Projede, Üniversite Bilgi Sistemleri için REST tabanlı geliştirilmiş mimariler, Temmuz 2021'de kullanılmaya başlanmıştır. Temmuz, Ağustos, Eylül, Ekim, Kasım, Aralık ve Ocak ayları için oluşturulan tüm servislere ait yanıtlama süreleri grafiksel olarak betimlenmektedir. Hangi ayların daha yoğun olduğu gibi detaylı raporlara ulaşılabilmesi ile beraber hızlı çalışan bir sistem sayesinde kağıt israfından da kaçınılmış bulunulmaktadır.

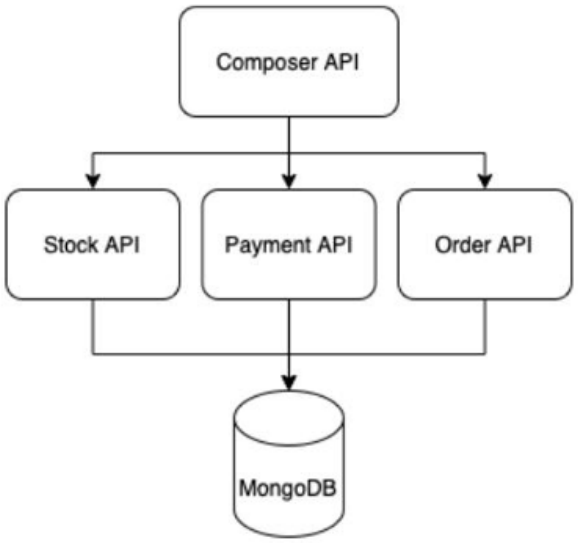


Şekil 3. Toplam Istek Sayısı ve Ortalama Yanıt Süresi

Kaynak: Altınkaya, C. (2022). *Üniversite bilgi sistemleri için REST tabanlı bir web servis platformunun tasarımı ve geliştirilmesi* [Yüksek lisans tezi]. Atatürk Üniversitesi.

Bu çalışma kapsamında, REST tabanlı mimarilerle Üniversite Bilgi Sistemleri için hızlı çalışan bir sistem oluşturulmuştur. Aylık yanıt süreleri grafiksel olarak gösterilmiştir. Detaylı olarak elde edilen kazanımlara yer verilmiştir.

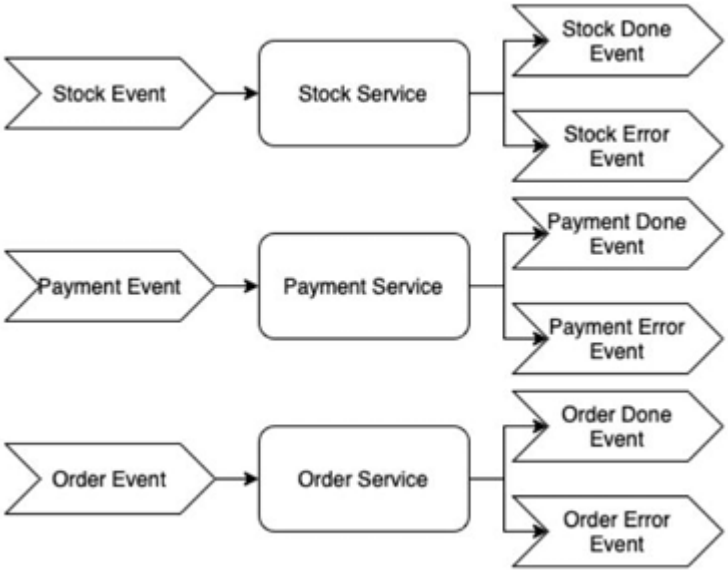
Gördesli ve Varol (2022) tarafından yapılan çalışmada, e-ticaret firmalarının mikroservis mimarisini tercih etmelerinin hata toleransı ve ölçeklenebilirlik olduğundan bahsetmektedir. Mikroservis mimari kullanımının zorluklarından olan servisler arası iletişim teknikleri, REST mimarisi ve asenkron iletişim, senkron iletişim tekniklerine ait performans analizi ve bu tekniklerin avantaj ve dezavantajlarını ele almışlardır. Bu çalışmada, e-ticaret modüllerinin temeli olan Stok, Sipariş, Ödeme süreçlerine odaklanılmıştır.



Şekil 4. Restful API'ler İle Eş Zamanlı Mimari Şeması

Kaynak: Gördesli, M. ve Varol, A. (2022). Comparing interservice communications of microservices for e-commerce industry. *2022 10th International Symposium on Digital Forensics and Security (ISDFS)*.

Sipariş ürünün satın alınmasından sonra stoktan düşürülür, ödeme oluşturulur ve sipariş veritabanına kayıt eklenir. Bu servisler arası iletişim sırasında hata alınması durumunda geri alma işlemi uygulanmaktadır. Tüm sipariş döngüsü 40 kere tekrarlanıp yaklaşık 3,04 saniye sürdüğü gözlemlenmiştir.



Şekil 5. Olay Odaklı Mimari İle Paralel Hizmet Tetikleme Şeması

Kaynak: Gördesli, M. ve Varol, A. (2022). Comparing interservice communications of microservices for e-commerce industry. *2022 10th International Symposium on Digital Forensics and Security (ISDFS)***.**

Sıralı yürütme durumunda tüm süreçlerin hata almadan tamamlandığı anlaşılmamaktadır. Sıralı yürütmeyle sipariş döngüsü 40 kere tekrarlanıp yaklaşık 3,03 saniye sürdüğü gözlemlenmiştir. Paralel yürütme uygulanarak tüketici tüm süreçlerin tamamlanmasının ardından sipariş oluşturabilmektedir. Hata alınması durumunda servisler arası hata tetiklenmesi ile önlenmeye çalışılmıştır. Bütün sipariş döngüsü 40 kere tekrar edilip yaklaşık 1 saniye sürdüğü gözlemlenmiştir.

Bu çalışma kapsamında, paralel yürütme ile sipariş sürecinin en hızlı şekilde oluşturulduğu sonucuna varılmıştır. Ancak, mikroservislerin hata durumlarını kontrol etmek ve sürecin tamamlandığına karar vermenin daha zor olduğu sonucuna ulaşılmıştır. Mimarilerin en önemli özelliği olan hataya karşı toleransın ve geri alma gibi eş zamanlı mimarilerin kullanımında daha kolay olduğu görülmüştür. Ayrıca, hangi mimarinin kullanılacağına tüketicilerin ihtiyaçları doğrultusunda karar verilmesi gerektiği belirtilmiştir.

Zhou vd., (2018) tarafından yapılan çalışmada, mikroservis mimarisinin endüstriyel alanda kullanımı sonucu ortaya çıkan hataların analizi ve sıklıkla yaşanan hataların geliştiricilere yaşattığı zorlukları öğrenmek amacıyla geniş çaplı bir çalışma ele almışlardır. Çalışma kapsamında, mikroservis mimarilerin tasarımı ve dağıtımı gibi konular dışında hata yönetimi üzerine çok az çalışma olduğu belirtilmiştir. Bu bağlamda, mikroservis mimarisi üzerine minimum 3 yıl, endüstriyel yazılım alanında ise minimum 6 yıl çalışmış olan 46 mühendis, belirlenen sorular dahilinde anket çalışması yapmak için davet edilmiştir. Kabul eden 16 mühendis arasından, genel kapsamı mikroservis mimarisini neden tercih ettikleri ve aldıkları hataları hatırlayarak sebeplerini ve hata ayıklama süreçlerini ele alan sorular yöneltilmiştir. Yanıtlanan soruların ortak sonucunda 22 adet hata durumuna ulaşılmıştır.

TrainTicket adını verdikleri projede bilet satışı, rezervasyon ve ödeme gibi konuları ayrı mikroservisler olarak ele alıyorlar. Anket sonucunda elde ettikleri 22 adet hata durumunu, projede oluşmalarını sağlayıp, oluşan hataları görselleştirebilecekleri programlar üzerinden izlenmektedirler. Yaşanan hata durumlarının görselleştirilmesinin projeye sağladığı katkılara da değinilmiştir. Yaşanan hata durumları fonksiyonel hatalar, işlevsel olmayan hatalar, dahili hatalar vb. kategorilere ayrılmıştır. Bu hataların mikroservis mimarisi özelinde mi, yoksa monolitik bir mimari tercih edilmesi durumunda mı yaşanıp yaşanmayacağı sorusuna da yanıt bulunmuştur.

Mimariler arasında kıyaslama yaparak, REST istemcisi kullanılmasının mikroservislerin geliştirilmesi konusunda kolaylık sağladığı belirtilmiştir. Servisler arası iletişimde hızın sağlanabilmesi gibi Docker vb. konteyner teknolojilerinin kullanılması, yük dengeleme, esneklik ve verimlilik açısından mikroservis mimarisinin kullanım kolaylıklarından bahsedilmiştir.

Anket sonuçları göz önünde bulundurularak, hata tespitinin nasıl yapılacağı, aşamalı olarak 7 adıma ayrılmıştır. Süreçler, hata alınması durumunda tekrar edilmektedir. Hata durumu seviyelere göre temel log, görsel log ve görsel işaret analizi şeklinde ayrılmıştır.

Maturity Level	Systems	Percentage
Basic Log Analysis	A1, A7, A11	23%
Visual Log Analysis	A2, A3, A8, A9, A10, A12	46%
Visual Trace Analysis	A4, A5, A6, A13	31%

Şekil 6. Hata Durumlarına Göre Analiz Şeması

Kaynak: Zhou vd. (2018). F*ault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study*. IEEE Transactions on Software Engineering,47(2), 243-260.

Yazılım geliştiriciler tarafından yaşanan hatalar için ayrılan zaman süreleri ve hatanın önem durumunun gösterilmesine de yer verilmiştir. Hata türlerinin gösterim şekilleri, 13 yazılım geliştiriciye sorularak fayda durumu analiz edilmiştir.

TrainTicket uygulamasında hata analizi iki bölüme ayrılmıştır. Birinci bölümde genel hatalara karşı alınan önlemler, 12 senaryo üzerinden ele alınmıştır. İkinci bölümde ise alınan hataların görselleştirilmesi ve izlenmesinin hatanın çözülmesindeki etkisi ele alınmıştır. Uygulama ve mikroservisler hakkında bilgi sahibi olan 6 yüksek lisans öğrencisi, hata durumlarının ayrıştırılması sonucunda hataların çözümlenmesi için geliştirilme yapmaktadır. Yaşanan hatanın çözümlenmesi için 2 saat süreleri bulunmaktadır. Hataların gösterilmesi için hata görselleştirme aracı kullanılmaktadır. Kullanılan görselleştirme aracına dair bir senaryo ele alınmıştır. Görselleştirme aracının kullanılması ile yaşanan hata durumu şekiller üzerinden detaylı olarak analiz edilmiştir.

Fault	Overall Time (H)	Time of Each Step (H)							Strategy	#N.	#E.	#UR.	#FR.	#FE.	Hit	Tactic
		IU	ES	FR	FI	FS	FL	FF								
F1	2	0.6	-	-	0.6	0.6	0.2	0.1	Service	7	228	4	1	8	Y	T3
F2	2.4	0.6	0.6	0.4	0.4	0.4	0.6	0.3	Service	21	2706	3	2	9	Y	T4
F3	failed	1	0.6	0.6	0.8	failed	failed	failed	Service	11	398	2	2	39	N	T3
F4	failed	0.8	0.8	0.3	1	failed	failed	failed	Service	22	1704	5	-	-	N	T3
F5	2.2	0.6	0.4	0.3	0.4	0.3	0.2	0.1	State	17	972	2	3	255	N	T1, T2
F7	2.6	0.8	-	-	0.8	0.6	0.3	0.2	Service	22	1705	3	3	24	Y	T1
F8	2.8	0.6	0.6	0.6	0.4	0.4	0.4	0.2	Service	7	178	4	1	4	Y	T1
F10	2.1	0.6	0.2	0.2	0.3	0.3	0.4	0.1	Service	16	960	6	2	28	Y	T1
F11	2.3	0.4	0.4	0.3	0.4	0.4	0.4	0.1	State	9	303	4	3	24	Y	T1, T2, T3
F12	1.3	0.3	0.2	0.2	0.4	0.1	0.1	0.1	State	9	210	5	3	4	Y	T1, T2
F13	1.6	0.3	0.2	0.2	0.3	0.3	0.2	0.2	Service	12	372	7	3	57	Y	T4
F16	2.3	0.3	0.3	0.3	0.4	0.4	0.4	0.2	Service	6	233	2	2	3	N	T3

Şekil 7. Geliştirilmiş İzleme Görselleştirilmesi ile Hata Durumu Şeması

Kaynak: Zhou vd. (2018). F*ault analysis and debugging of microservice systems: industrial survey, benchmark system, and empirical study*. IEEE Transactions on Software Engineering,47(2), 243-260.

Bu çalışma kapsamında, mikroservislerin kullanımı sırasında oluşan hataların endüstriyel uygulamalarda tespit edilmesi üzerine geniş çaplı araştırmalara yer verilmiştir. Endüstriyel uygulamaların hata yönetimi konusu derinlemesine incelenmiştir. Bu konu üzerinde araştırmaları olan mühendisler de çalışmaya anket soruları kapsamında dahil edilmiştir. Yapılan anket çalışması sonucundaki bulgulara dayanarak oluşturulmuş olan TrainTicket uygulamasına yer verilmiştir. TrainTicket uygulaması üzerinde oluşan hata loglarının görselleştirme teknikleri, çözüm mekanizmaları ve hata izlenmesi gibi konularda orta ölçekli projeler için aydınlatıcı olmaktadır. Büyük ve karmaşıklık seviyesi yüksek olan endüstriyel uygulama mikroservislerinin hata görselleştirme konusunda yapılabilecek geliştirmeler detaylı bir şekilde incelenmiştir. Ayrıca büyük endüstriyel uygulamalarda monolitik mimariden mikroservis mimarisine geçilmesinin gerekliliklerinden ve karmaşıklığın da az olduğu monolitik mimarinin yerine mikroservis mimarisinin tercih edilme sebeplerinden de bahsedilmiştir.

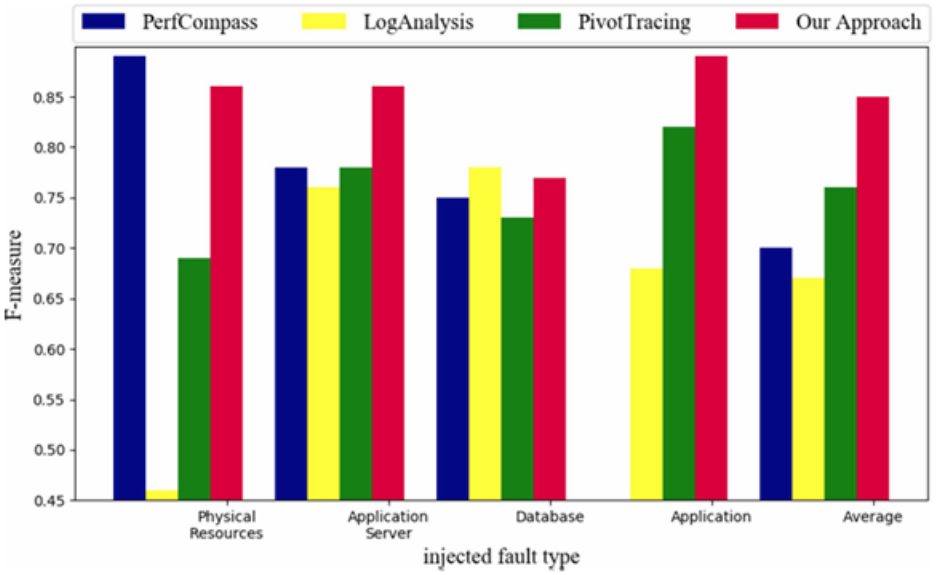
Wang, Zhang, Xu ve Gu, (2020) tarafından yapılan çalışmada, mikroservis mimarisinde yazılmış olan projelerde yaşanan hataları otomatik olarak tespit etmek için istatistiklere dayalı hata teşhisi üzerine bir çalışma yapılmıştır. Mikroservis tabanlı uygulamalarda (Google, Netflix vb.) büyük kitlelere hitap edilmesi ile birlikte karmaşıklık seviyesi artmıştır. Bu nedenle, karmaşıklık seviyesi yüksek uygulamalarda hata tespitinin zor olduğu belirtilmiştir. Hata tespitleri çalışma sırasında üç kategoriye ayrılarak incelenmiştir. Akıllı hata teşhisi; mikroservislerde yaşanan hata durumlarının manuel müdahale ile uzun sürmesi ve yoğun emek gerektirmesi nedeniyle otomatik olarak teşhis edilmesi durumudur. Çevrimiçi hata teşhisi; projede geliştirilen mikroservisin güncellenmesi sonucunda oluşan hatalar gösterilmemektedir. Mikroservislerin etkileşimleri sonucunda oluşan hataları kümeleme yöntemi ile dinamik olarak teşhis etme durumudur. İş akışına duyarlı hata teşhisi; çeşitli mikroservisleri içeren sistemlerde, örneğin, e-ticaret uygulamalarında farklı görevleri olan mikroservislerin iş akışındaki farklılıkların teşhis edilme durumudur.

İş akışına duyarlı otomatik hata teşhisi ile hataların toplanması, sınıflandırılması ve performans farklılıklarındaki hataların teşhis edilmesi sağlanmaktadır. Ağaç yapısı kullanılarak manuel müdahaleye ihtiyaç olmadan otomatik hata teşhisi sağlanmaktadır. Mikroservisler arası iletişim ağaç yapısı ile gösterilmektedir. Gerçekleştirilen isteklere ait benzer içerikler kümelere ayrılır. İş akışındaki anormallikler tespit edilir. Anormallikleri tespiti etmek için yazılmış olan algoritmada, ilk tespit edilen hata olasılığı en yüksek kabul edilir ve olası tüm ihtimaller, çağrılma sırasına göre sıralanmaktadır.

Mikroservislerde, benzer iş akışına sahip olan servislerin performansı, aynı isteklere yaklaşık olarak aynı sürelerde yanıt vermesi gerektiğinden bahsedilmektedir. Yapılan işlem sayıları, sunucunun darboğaza ulaşmadığı sürece sabit olduğundan bahsedilmiş ve servislerde istikrarsızlık olmaması durumunda, yaşanan durumun önemini anlamak için belirli kat sayılara yer verilmiştir.

Yapılan araştırmalar, mikroservis mimarisini kullanan SockShop, PiggyMetrics ve TrainTicket projeleri üzerinde iş akışına ve performans durumuna bağlı olarak değerlendirilmektedir. Ağ üzerinde yaşanan paket kayıpları, veritabanının yanlış

yapılandırılması ve yapılan istek sonucu yanıt sürecinde yaşanan gecikme durumu detaylı bir şekilde incelenmiştir. Kullanılan yöntemin doğruluğunu görebilmek adına PerfCompass, PivotTracing ve LogAnalytics yöntemleri ile analiz edilmiştir.



Şekil 8. Hata Durumu Karşılaştırma Grafiği

Kaynak: Wang, T., Zhang, W., Xu, J. ve Gu, Z. (2020). W*orkflow-aware automatic fault diagnosis for microservice-based applications with statics*. IEEE Transactions on Network and Service Management *17*(4), 2350-2363.

Bu çalışma kapsamında, mikroservis mimarisinde yaşanan hataları otomatik olarak tespit edilmesi için istatistiklere dayalı olarak hata teşhisi üzerine çalışma yapılmıştır. Hatalar, ağaç yapıları üzerinde ayrıştırılmış ve bu ağaçlar aracılığıyla hataya sebep olan mikroservisler belirlenmiştir. Ayrıca, performans sorunları üzerine yaşanan hatalar içinde çalışma yapılmış olup, elde edilen bulguları test etmek amacıyla yapılan deneylere yer verilmiştir. İş akışı ve performans sorunlarıyla ilgili hataları mevcut deneylerde teşhis edebilmekle birlikte, işletim sisteminde yaşanacak olan hata durumlarını doğru bir şekilde tespit edilmemektedir.

Asrowardi, Putra, S, ve Subyantoro (2020) çalışmasında, e-ticaret üzerine yazılan uygulamalarda mikroservis mimari tasarımının nasıl olması gerektiği sipariş oluşturulma süreci üzerinden ele alınmıştır. Monolitik mimari yerine mikroservis mimarisinin tercih edilmesinin performans üzerindeki etkileri incelenmiştir.

E-ticaret firmalarının, internetin günden güne dünya genelinde daha yaygın kullanılmasıyla birlikte artan iş fırsatları sebebiyle müşteri memnuniyetini artırmak için çeşitli hizmetlere sahip olması beklenmektedir. Uygulamalar geliştirilirken mikrosevis mimarisinin kullanım sebeplerinin başında gelen her bir mikroservisin kendi görevininin olması ile birlikte, RESTAPI gibi dil bağımsız bir şekilde birden çok platformda kullanım imkanı sağlayan web hizmetleri kullanılmaktadır. Yazılımlar geliştirilirken günümüzde üst düzey programlama dili olarak ele aldığımız Nesne Yönelimli Programlama (OOP) sıklıkla tercih edilmektedir.

Çalışmada seçilmiş olan mimarinin tercih edilmesinin en önemli sebebi, geliştirme sürecini kolaylaştırılmasıdır. Yaşanan hatalardan kaynaklanan sorunları ele almaktadır. Sektördeki büyüme ile beraber güvenlik ve performans üzerinde yaşanan sorunları kolaylaştırmaktadır.

Performance	Microservices Times	Monolithic
Loading	283.6 ms	175.0 ms
Scripting	1180.4 ms	1705.0 ms
Rendering	479.0 ms	377.7 ms
Painting	45.2 ms	47.1 ms
System	790.0 ms	471.5 ms
Accessibility	66	66
Best Practice	79	79
SEO	91	91

Şekil 9. Performans Test Sonuçları

Kaynak: Asrowardi, I., Putra, S, D. ve Subyantoro, E. (2020). Designing microservice architectures for scalability and reliability in e-commerce. *Journal of Physics: Conference Series (Vol. 1450, No. 1, p. 012077).*

Mikroservislerin kullanıldığı gerekli API'ler üzerinde tüketicilerin en iyi performans elde edebilmesi için hata durumu, güvenlik ve hız gibi konular üzerinde belirli tekrarlarla test yapılmıştır. Yoğun kullanım altında yapılan bu testin sonuçlarına da yer verilmiştir.

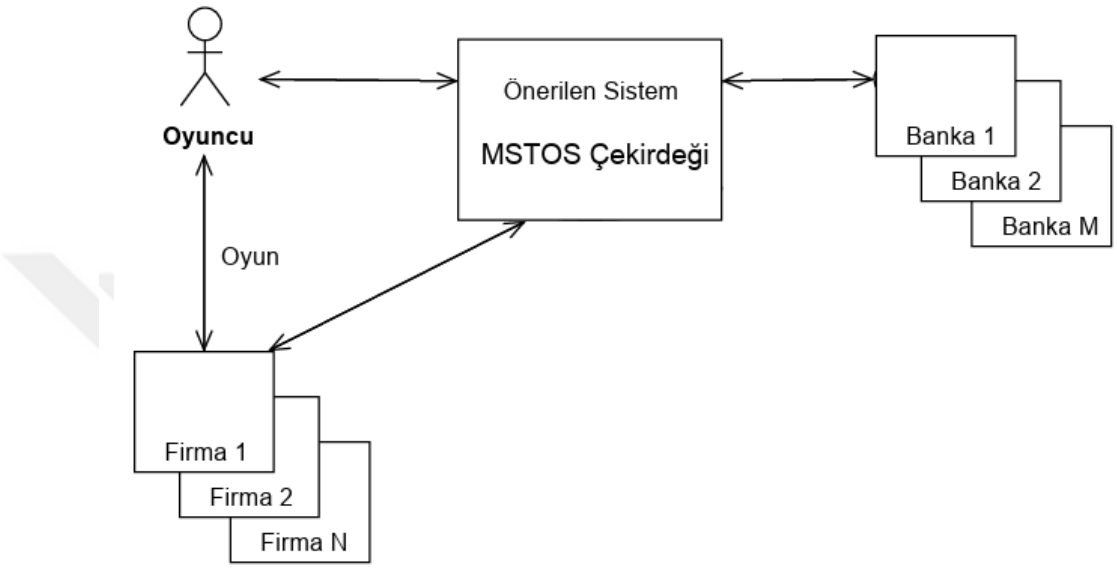
Bu çalışma kapsamında, e-ticaret uygulamalarında tercih edilen mikroservis mimarisinin sipariş sürecinde 300, 500 ve 1000 tekrar üzerinde 22.000 veri ile performans testi yapılmış, ve test sonuçlarına grafikler halinde yer verilmiştir. Mikroservis mimari kullanımının monolitik mimariye göre daha performanslı olduğu sonucu elde edilmiştir.

Kocaman (2018) tarafından yapılan çalışmada, kitlelere hitap eden oyun uygulamaları yerine daha çok popülerlik kazanmaya çalışan oyun uygulamalarına güven sağlanması adına mikroservis mimarinin getirmiş olduğu özellikleri kullanarak ödeme sistemi tasarlanmıştır. Bu ödeme sistemi iki yönlü olarak çalışmada ele alınmaktadır.

Çalışma, öncelikle firmalar üzerinde yapılabilecek anlaşmalar dahilinde, kullanıcının MSTOS projesine güvenilir bir şekilde bakması ve kart bilgileri gibi önemli bilgileri, oynamak istediği oyun için yapması gereken ödemeyi MSTOS üzerinden yapmasını amaçlamaktadır. Proje kapsamında avans verme ve eksi bakiyeye düşülmesi gibi içerikler de detaylandırılmıştır. Projenin geliri ise oyun firmaları ile yapılan anlaşmalar kapsamında belirli bir komisyon ile alınacağı belirtilmiştir.

Oyuncular için çalışma üzerinde hızlı şekilde ödeme gerçekleştirmek adına oyun içerisinden çıkmadan bakiye yüklemesi sağlanabilmektedir. Fakat kart ile yapılan ödeme banka sisteminde öncelikle provizyonda olacağından dolayı oyun bitiminde provizyonun iptal edilmesi ile MSTOS zarar edebilir, fakat bu durum tolere edilmiştir. Oyuncular, hesaplarında kalan bakiyeyi istedikleri takdirde başka oyunculara aktara bilmektedir. Oyun sırasında yapılan ödemeler jeton olarak alınıp, alınan jetonlar firma üzerinde puan şeklinde verilmektedir. Kullanıcılar ödemelerini tek firmaya yaptıkları için firma tarafında kullanımın artırılması adına avans verilmesi gibi öneriler de içermektedir. Bir firmanın sahip olduğu birden fazla oyun türü olabilir, fakat hepsi sadece jeton sistemi ile kolayca entegre olabilmektedir. Ayrıca mevcut bulunan jetonlar MSTOS sistemini kullanan tüm firmalarda ortak olarak oyuncular tarafından kullanılabilmektedir. Sıklıkla tercih edilen

oyunlara yönelik ödeme sistemlerinden de bahsedilmekte olup mevcut sistemler gibi anlaşılabilirliği yüksek bir sistem tasarlanmaktadır.



Şekil 10. Önerilen Sistem Mimarisi

Kaynak: Kocaman, Y. (2018). M*ikroservis tabanlı ödeme sistemi tasarımı ve gerçekleştirilmesi* [Yüksek lisans tezi]. Yeditepe Üniversitesi.

MSTOS, oyuncu ve firmalar arasında ödeme sisteminin sağlanması adına köprü görevi görmektedir. Mikroservis mimarisi kullanılarak hazırlanmış olan ödeme sisteminde oyuncu sayısındaki artış sırasında hata yaşanmaması için ölçeklenebilirlik problemi göz önünde bulundurulmuştur.

Uygulama, firmalar ve oyun arayüzü olarak iki kısımda yazılmıştır. Arayüzde HTML, yazılım tarafında ise Hipermetin Önişlemcisi (PHP) dili tercih edilmiştir. Tek kullanımlık oluşturulan şifre ile uygulamaya giriş sağlanmaktadır. Yazılım kütüphanelerinin kullanılması ile uygulama içerisinde jeton satın alma işlemi sağlanır. Oyun içerisinde hızlı satin alma ile oyundan çıkmadan sadece bir jeton satın alınmaktadır.

MSTOS projesinde oluşturulan mikroservisler için ana sayfa, kullanıcı girişi, satın alma, hızlı satın alma, jeton sorgulama ve jeton harcamadan oluşan istekler 60 saniye boyunca tekrar tekrar sunucuya gönderilerek ölçeklendirilme testi yapılmıştır. Test

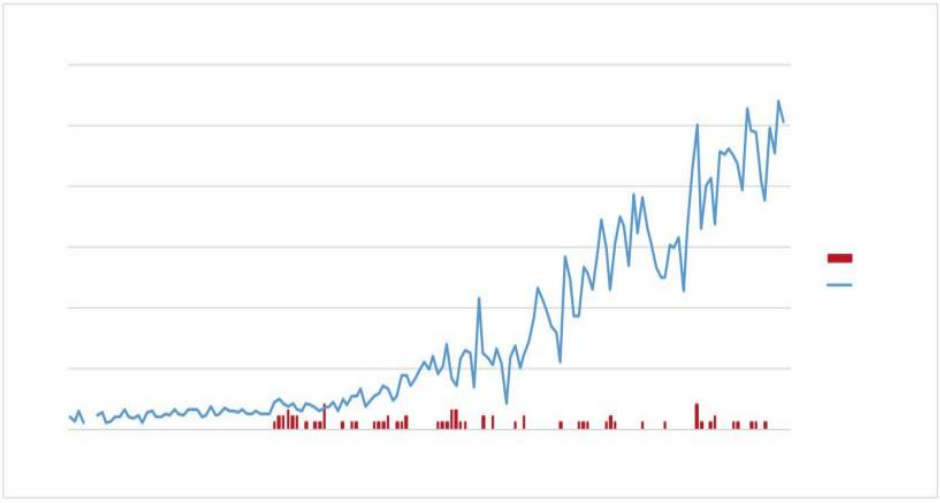
kapsamında sunucu adedi 10 binden istikrarlı olarak 50 bine kadar çıkarılmış ve konteyner adedindeki yatay ölçeklendirme ile beraber yetersiz kalındığı sonucuna ulaşılmıştır. Sunucu sayısı arttırılıp kullanıcı sayısı 30 bin ila 50 bin arasında istikrarlı olarak artırılarak ölçeklendirme durumu test edilmiştir. Testin başarı durumunun belirlenmesi için kriter olarak hata alınması değerlendirilmiştir. Toplam yapılan istek oranı ile hata alınmamış isteklerden yanıt beklenilmesi ile başarı oranı hesaplaması yapılmıştır. Yapılan bu performans analizi tablo olarak belirtilmiş ve hata dönmeyen istekler başarılı olarak belirtilmiştir. Sunucular üzerinde yapılan testler detaylı şekilde grafikler halinde belirtilmiştir.

Bu çalışma kapsamında, mikroservis mimarisi ile tasarlanan ödeme sistemi üzerindeki ölçeklenebilirlik detaylı bir şekilde analiz edilmiştir. Çalışmanın yapılmış olduğu yazılım geliştirme araçları ve yazılım geliştirme dilleri hakkında detaylı bilgi verilmiştir. Kullanılan araçlar ve dilin, proje için verimli olacak şekilde seçildiği belirtilmektedir. Oyun için ödeme yapılması sırasında çok fazla sayıda kullanıcının istekte bulunması durumunda, gelen istekleri karşılama konusunda yaşanan yetersizlikler üzerine yapılan testlerin analizleri grafiklerle desteklenmiştir. Uygulamadan kaynaklanan hataların ölçeklendirme problem ile doğrudan bağlantılı olduğu sonucuna varılmıştır. Hata durumlarının azaltılması amacıyla uygulamadaki isteklerin kullanıcı sayısının artmasıyla, konteyner kullanımı ile doğru ölçeklendirme seçiminin yapıldığı durumda sorunun çözüldüğü görülmüştür.

Hasselbring ve Steinacker (2017) tarafından yapılan çalışmada, monolitik mimari ile yazılmış olan otto.de e-ticaret sitesinin günümüzde ulaştığı popülerlik sebebiyle, kullanıcılara daha iyi hizmet sunabilmek amacıyla mikroservis mimarisini tercih etmelerinin gerekliliklerinden ve getirdiği kazanımlardan bahsetmektedir. Yüksek seviyede güvenilir, ölçeklenebilir ve yüksek hata toleransına sahip olan uygulamaların temelinde mikroservis mimari yapısı kullanılması sebebiyle otto.de üzerinde geliştirme aşamaları ele alınmıştır. Dikey olarak tasarlanmış mimari ile öncelikle ürün, sipariş, promosyon ve arama modülleri geliştirilmiştir. İlerleyen yıllarda mimariler daha da

özelleştirilerek, sadece erişim sağlayan tüketici bilgilerini paylaşan servisler arasında ortak olarak kullanılmaktadır.

Mikroservis tabanlı mimarinin kullanımıyle otto.de e-ticaret sitesinde yapılan geliştirmeler, servis bazında ekiplere bölünmüş ve zamanla canlı ortamdaki güvenilirlik iyileştirilmiştir.



Şekil 11. 2 Yıl İçerisinde Otto.de'de Dağıtım Ve Canlı Olayları Şeması

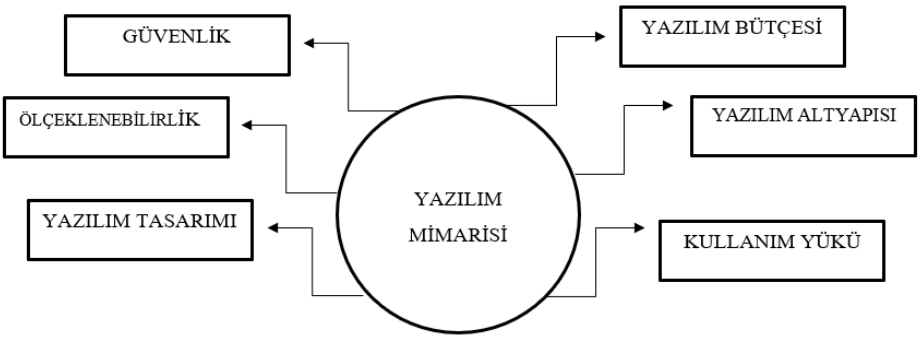
Kaynak: Hasselbring, W. ve Steinacker, G. (2017). Microservice architectures for scalability, agility and reliability in e-commerce. *2017 International Conference on Software Architecture Workshops (ICSAW).*

Bu çalışma kapsamında, dağıtılmış bir şekilde oluştulmuş mikroservis mimarisinin, monolitik mimariye göre daha karmaşık olduğu ancak tutarlılığın sürdürülmesi, alarm mekanizması ve hata toleransı gibi e-ticaret projelerinde önem arz eden konular için geniş bir operasyon ekibiyle çalışılması gerektiği belirtilmiştir. Yazılım üzerinde başarıyla uygulanan ölçeklendirmeler sonucunda arıza ve hatanın erken tespit edilebilir hale gelmiş ve geliştirme süresinde azalma yaşandığına değinilmiştir.

2.YAZILIM MİMARİLERİ

Yazılım mimarisi, bir yazılımın unsurlarının, düzenlenmesini, sistemde bulunan parçaların birbirleriyle nasıl ilişkileri ve yapılar ile bağlantı kuracağını açıklamaktadır. Her proje kendine özgü özelliklere sahiptir. Bu özellikler göz önünde bulundurularak hangi mimarinin seçileceği değişen ihtiyaçlara bağlı olarak seçilmelidir. Yazılım projelerindeki karmaşıklığı yönetmek, performans sorunları, güvenlik ve uzun vadeli bakımını sağlamak gibi kritik konular üzerinde planlama yapılması gerekir. Projenin belli bir sistematik üzerinde ortaya çıkmasını planlamak üzere yazılım mimarilerine ihtiyaç duyarız. Projeler iyi hazırlanmış yazılım mimarilerine sahip olmaması durumunda teknolojinin gelişmesiyle beraber yeni yazılımlar ve geliştirilmeleri kabul etmez. Yeni teknolojilere ayak uydurmayan yazılımlar, maliyetleri arttırabilir, güvenlik zafiyeti yaşayabilir ve sadık müşteri tabanının kaybedilmesine yol açabilir.

Yazılım mimarisi genellikle projenin ilk tasarım aşamasında oluşturulur. Bu aşamada, mimari tasarım, performans ve güvenlik gibi kalite sorunlarını ele alır. Ayrıca, mimarlık, kodlama ve bakım gibi diğer geliştirme faaliyetleri için bir referans noktası sağlar (Galster ve Angelov, 2016). Yazılım mimarisi, yazılımcının her anında bulunur. Büyük veya küçük yazılım projelerinin kapsamı farketmeksizin önemlidir. Süreç içerisinde verilmiş olan kritik kararların tamamını içermektedir. Yazılım geliştirici için mimariyi oluşturmak en temel unsurdur. Evrimsel mimari olarak adlandırılan kavram, yazılım ile beraber mimarinin inşa edilmesini ifade eder. Günümüzde yazılım projelerinde Çevik olarak mimari yaklaşımların öncelikleri belirlenir, süreç içerisinde değiştirilecek kararlara da uygun olarak bir yaklaşım seçilmektedir. 2001 yılından bu yana Çevik metotu benimsemeye çalışan kuruluşların sayısı artmıştır. Çeviklik; önceden öngörülemeyen, belirlenemeyen koşullarda uygulanabilecek genel bir strateji olarak tanımlanmıştır.



Şekil 12. Yazılım Mimarisi Gereklilikleri

Kaynak: Yazar tarafından oluşturulmuştur.

Yazılım mimarisine dair olan arayışın sebebi, yazılımcıların programın akışı içerisinde oluşan farklılıkların değiştirilemeyeceğinin düşünülmesidir. Mimari üzerinden verilmiş olan kararların, projenin ilerleyen zamanlarında gereksinimlerinin değiştirilebilmesi gerekebilir. Yazılım mimarları projenin ilerleyişiyle birlikte ortaya çıkan gereksinimlere göre yazılımın ihtiyaç duyduğu, hem yazılım dili hem de veritabanı seçimi gibi konularda karar verici kişidir. Günümüzde bu kişiler belirli seviyelere ulaşmış olan kişiler olsalar da, gün geçtikçe hayatımıza giren mikroservis yapısı ile beraber bağımsız olarak kendine özgü mimarisi olan uygulamaların geliştirilmesine katkı sağlanmaya başlanmıştır.

Kurumsal mimari gibi yazılım yapıları kendi içerisinde uygulama mimarisi, çözüm mimarisi gibi çeşitleri bulunmaktadır. Teknik anlamda donanımı yeterli olan mimariler iş ile ilgili içeriklere göre seçilir. Teknik ve işin mevcut gereksinimlerine uygun mimari seçimi yapılması kritiktir. Mimariye dair yapılması gerekenler, iş ile ilgili içeriklere göre alınan geri bildirimlerle geliştirilmesi sağlanır. Performans, güvenlik, iş gereksinimleri ve ölçeklenebilirlik gibi faktörlere dayanarak mimari seçimi yapılması gerekmektedir. İşin mevcut gereksinimlerinin bakım kolaylığı, ölçeklenebilirlik gibi teknik konuların dışında kullanıcının ihtiyaçları ve iş hedeflerinin de gözetilmesi gerekmektedir.

Pandemi döneminde hayatımıza giren hibrit çalışma sistemi ile beraber zamandan sağlanan tasarruf, yazılım ekiplerinin iş dağılımında farklılıklara neden olmaya

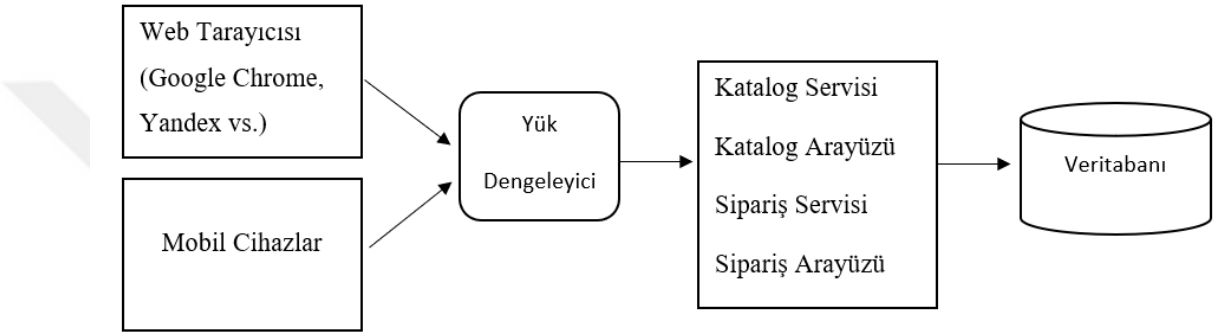
başlamıştır. Birçok yazılım geliştirme ekibi, farklı coğrafi konumlarda çalışan ekip üyelerinden oluşmaktadır. Sadece yazılım geliştirilmesi için değil, farklı zaman dilimlerinde bulunan ekibin de projeye katkı sağlaması açısından mimarinin sağlam temeller üzerine oluşturulması, yazılım kalitesini de artırmaktır. Yazılım mimarisini oluştururken sadece yazılım değil, ekip içerisinde de yazılım mimarisine dair katkı sunulması daha kıymetli hale gelmiştir. Kullanılacak olan kütüphane, veritabanı gibi içerikleri işi yapan yazılımcılara seçtirerek oluşturulacak olan mimariden minimum hata ve maksimum verim ile ilerlenmesi muhtemeldir. Ayrıca, ekip içerisindeki iletişim ve yetkinlik seviyesi de mimarinin oluşturulması aşamasında, hataların azaltılması, verimliliğin artmasına sebep olması muhtemeldir.

Geçmişten günümüze yaşanan gelişmelerle beraber konteyner yapıları kullanımıyla mimariye herkesin geçmişe nazaran daha etkin şekilde katkı sunacağı hale gelmiştir. Yazılımsal anlamda geniş kitlelere hitap eden büyük yazılımcıların daha sabit ve değişmeyen kararları bulunmaktaydı, fakat günümüzde daha küçük yapılar şeklinde ayrı yazılım mimari yorumlamalarının yapılmasıyla her statüden yazılımcının geliştirmesine açık biçimdedir. Yazılım geliştirme sürecinin kritik aşaması olan kod yazılmaya başlanılması sürecinde yazılım mimarisi seçilmeye başlanmış demektir. Çünkü, projenin ilerlemesi ile beraber yapılan geliştirmelerde mimarinin temellerinin üzerine katılarak ilerlenmektedir. Teknolojinin gelişmesi ile beraber yazılımlar gerekli olan servisler ile entegrasyonlara açık halde geliştirilmektedir.

Uygun tasarlanmamış olan yazılım mimarisi her açıdan zarar vermektedir. Örneğin, kullanıcı etkileşiminin arttığı durumlarda yavaşlayan uygulamalar mimari açıdan uygun değildir. Yazılım mimarisi, sınıf seviyesinde yapılacaklara karışmaz, fakat genel olarak sistemin nasıl işleyeceğine, içerisinde bulunan bileşenlere ve nasıl haberleşeceklerine karar verir. Yazılım mimarisinin gerçek projelerde uygulanmaması durumunda sonuçları net biçimde ortaya konulamaz. Bu sebeple, büyük çaplı projelerde mikroservis mimari mantığı ile parçala, böl ve yönet şeklinde ilerlemek daha kullanışlı hale gelmiştir.

2.1. Monolitik Mimari

Geçmiş yıllarda yazılım geliştiriciler, Monolitik Mimariyi uzun yıllar kullandılar. Farklı bileşenlere örnek olarak iş mantığı, bildirim modülü ve yetkilendirme gibi bölümleri tek bir programda birleştirdiler. Tüm yazılım geliştirme süreci boyunca uygulama tek bir bütün olarak dağıtılır (Gos ve Zabierowski, 2020).



Şekil 13. Monolitik Mimari Şeması

Kaynak: Yazar tarafından oluşturulmuştur.

Monolitik uygulamalar üç farklı mimari katmandan oluşmaktadır: Veritabanı, iş mantığı ve kullanıcı arayüzü seviyesi olarak belirlenmiştir. İngilizce ve Fransızca dillerinde "mono" tek anlamına gelmektedir. Monolitik kelimesi büyük programları tanımlamak için kullanılmıştır, teknolojinin gelişmesiyle beraber bakımı giderek zorlaşmıştır (Kyryk vd., 2022). Küçük ve basit uygulamalar, hızlı şekilde kullanıcıya sunulması gereken başlangıç seviye projeler, çok katmanlı mimarilere ihtiyaç duymayan kişisel projelerin geliştirilmesinde monolitik mimari kullanımı daha uygundur. Üç katman halinde gördüğümüz monolitik mimari, veri katmanında uygulamanın sahip olduğu verilerin sistem üzerinde belirli bir düzen içerisinde saklanmasına ve yönetilmesine olanak sağlar. Veritabanları (MongoDB, MySQL, NoSQL, Redis, Cassandra) gibi tercih edilebilir. Veritabanı tercihi de mimari tercihi gibi uygulama içerisinde bulunacak verinin yük miktarı, ölçeklenebilirlik, esneklik, erişilebilirlik ve normalizasyon gibi önemli konular dikkate alınarak seçilmelidir. İş katmanı, uygulamanın iş mantığını içermektedir. Gelen istekleri alarak, veritabanından veri alır veya veritabanına veri gönderir. İş katmanı,

veri doğrulaması ile beraber hata işleme, güvenlik gibi uygulamanın temel işlevselliğine dair işleyişi ayakta tutan katmandır. Kullanıcı arayüzü katmanı, kullanıcıların uygulama içerisinde iletişim kurmasını sağlayan katmandır. Diğer iki katman ile sürekli iletişim halindedir. Kullanıcının isteklerini sunucuya iletir, veri akışı sağlandığında arayüzde gösterilmesini sağlar. Kullanıcıların isteklerine göre karşılaşılan hatalar sonucunda hata mesajları ile kullanıcıya dönüş yapar. Kullanıcının menü, form veya buton gibi arayüzünün görsel tasarımını sunar. Bu katmanların temel amacı, yazılım geliştirme sürecinin bölünmesiyle daha kolay ve bakımı rahat bir şekilde yapılabilecek yazılımlar elde etmektir. Monolitik mimari, anlaşılırlığı yüksek, mimari tasarımı kolay olması ile beraber projeler'de tercih edilme sebepleri farklılık göstermektedir.

Monolitik mimariler anlaşılması, geliştirilmesi, teknolojinin gelişmesi ile beraber zaman içerisinde bakımı zor olan sistemlere dönüşmektedir. Bu tür durumlarda yazılım geliştiriciler uzun saatler boyunca zaman harcaması yapması gerekmektedir. Geliştirilmiş olan uygulamanın zaman içerisinde yeniden mimarisini inşa etmek ve modüler yapısını geri yüklenmesi gereken durumlar oluşabilir. Çünkü modüler yapısını eski haline döndürebilmek için yeniden mimari oluşturulması gerekmektedir. Genellikle yeniden mimari oluşturulma süreci geçici olarak yapılır (Terra, Valente, Bigonha, 2012).

Monolitik bir mimari de, tüm işlevsellik tek bir uygulama içerisinde kapsüllenmiş durumdadır, bu da modülleri bağımsız olarak çalıştıramadığınız anlamına gelmektedir. Mimari birbirine sıkı bir şekilde bağlı çalışmaktadır. Uygulama içerisinde yapılmış olan bir istek tüm mantıksal fonksiyonlar ile beraber tek bir işlem olacak şekilde çalışır. Bu durum programlama dillerinin temel özelliklerinden faydalanmanıza olanak sağlar. Geliştirmiş olduğunuz uygulamayı sınıflara, fonksiyonlara, isimlere bölünerek kullanılmasını sağlar. Uygulama içerisinde test edebilirsiniz. Yapılan değişiklikler çok kez test ederek yatak olarak ölçeklendirebilirsiniz. Mimariyi kullanarak projeye başlamak iyi bir fikir olabilir fakat sistemin karmaşıklığı ve sınırları açısından beraberinde getirdiği zorluklarda bulunmaktadır. Uygulama büyüdükçe, geliştirici sayısınında çoğalmasına bağlı olarak mimarinin dezavantajları da beraberinde gelmektedir (Ponce, Márquez, Astudillo, 2019).

2.1.1. Monolitik Mimarinin Avantajları

- Sistem Boyutu ve Karmaşıklığı

Uygulama küçük çaplı ve zaman içerisinde değişmeyecekse monolitik mimari kullanılması daha uygundur. Servisler arası iletişim kurma, dağıtım ve izleme gerektirdiğinden maliyet ve zaman açısından tercih edilmektedir. Tek bir uygulama olarak ele alındığı için dağıtımı daha kolaydır. Yazılım geliştirmek az sayıda çalışan ekipler için kolaylık sağlamaktadır. Bazı uzmanlara göre; sisteminizde yaklaşık 60'tan az kişi çalışıyorsa mikroservis mimarisine ihtiyaç duyulmamaktadır (Singleton, 2016).

- Yeniden Kullanılabilirlik

Monolitik bir mimari de oluşturulmuş olan uygulamanın kodun kendi içerisinde yeniden kullanımı daha kolaydır. Çünkü tüm bileşenler bir yapı içerisinde olduğu için basitlik sunmaktadır. Bileşenler birbirlerine doğrudan erişim sağlayabildikleri için veri paylaşımı kolaydır. Uygulama içerisinde sorun giderme ve test etme gibi konularda kolaylık sağlamaktadır.

- Performans

Monolitik mimari, uygulamanın bileşenlerinin tek bir paket içerisinde çalışmasına olanak sağlamaktadır, bu sayede iletişim gecikmesi minimum seviyeye indirilmiştir. Farklı bir ağ trafiği ya da veritabanı sorgusu içerisinde çağrı yapma ihtiyacı ortadan kalkmaktadır. Verilerin büyüklüğü arttıkça performans sorunlarına yol açacağı için ciddi derecede avantaj sağlamaktadır. Uygulama içerisinde hata ayıklama ve izleme tek bir kod tabanı içerisinde sağlandığı için problemlerin çözüm süreleri kısalır ve sistemin performansında artış sağlanır. Yazılım geliştirme sürecindeki kısaltma ile ayrıca ekiplerin performansında da artış sağlamaktadır.

2.1.2. Monolitik Mimarinin Dezavantajları

- Esneklik ve Ölçeklenebilirlik

Sanal sunucular ve sunucu üzerinde bulunan yükün dengelenmesi için ölçeklenebilirlik önemlidir. Uygulama geliştirilirken daha fazla yazılım kaynağı, zaman, lisanslama maliyetini göz önünde bulundurarak alternatif bir seçenektir. Fakat monolitik mimari ile geliştirilmiş olan uygulamalar da en çok kullanılan modülün ölçeklendirilmesi mümkün olmadığından dolayı tüm mimarinin ölçeklendirilmesi gerekmektedir. Bu duruma bağlı olarak işgücü ve maliyet artışı yaşanmaktadır.

Bir monolitik mimariyi ölçeklendirmenin dezavantajı, işlevselliğinin yalnızca bir kısmının en çok ne zaman kullanıldığına bakılmaksızın uygulamasının tamamının ölçeklendirilmesidir (Velepucha ve Flores, 2023).

- Sistem Kararlılığındaki Zayıflıklar

Monolitik mimari ile oluşturulmuş olan uygulamalar da bir modülün başarısız olması, hata döndürmesi, bakım gerektirmesi gibi sorunlarda sistemin kullanılamaz hale gelmesi demektir. Bu durum karmaşık olarak uygulanmış monolitik mimari de sorunlara yol açmaktadır. Sistemin kararlılığı ve dayanıklılığı sonucunda tüm uygulama etkilenmektedir. Tüm modüller tek bir kod tabanında birleştiği için hata tespitinin yapılmasında da zorluklar yaşanmaktadır. Örneğin, monolitik mimari ile oluşturulmuş olan bir e-ticaret projesinde ürünlere baktığınız sırada aslında ödeme modülünde sorun olmasına rağmen sistem komple etkilendiği için ürünler de ekranda görünmeyecektir. Veritabanı, sistem yoğunluğu ve hatalar çalışan modülün tespitinin yapılamaması uygulamalar için ciddi derecede kayıp demektir.

- Dil Kısıtlılığı

Monolitik mimari ile oluşturulan uygulamalar tek bir programlama dili ile uygulama yazıldığı için esneklik söz konusu değildir. Farklı teknolojilerin kullanılması projeye entegrasyon açısından sorunlara yol açabilmektedir. Gelişen teknolojiler ile beraber uygun teknolojinin seçimi de önem arz etmektedir, güncellemeler ile beraber

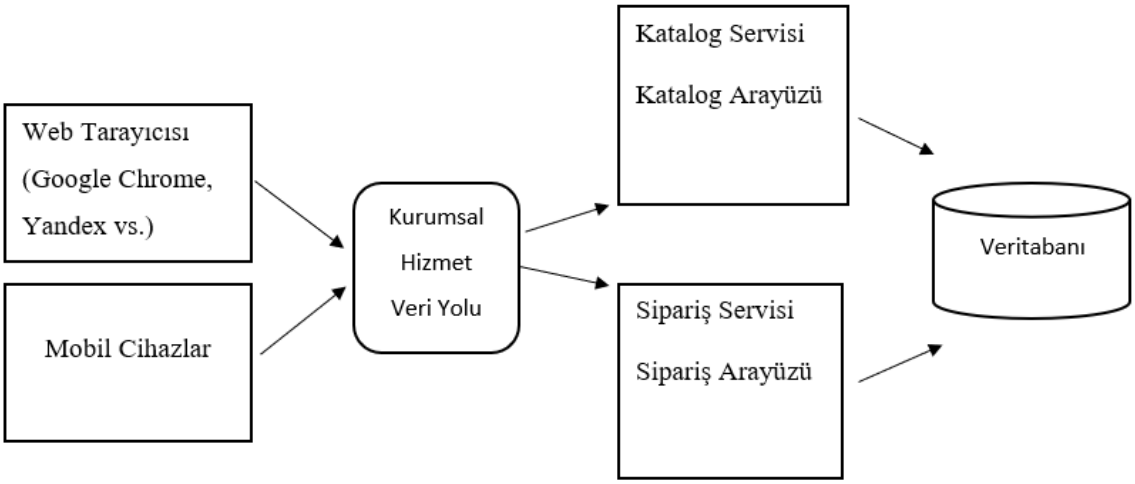
geçmiş versiyonların desteklenmemesiyle tek bir dile bağlı kalınması sorunlara yol açmaktadır.

- Koordinasyon Zorlukları

Monolitik mimari, ekiplerin birbirlerinden bağımsız olarak çalışmalarını zorlaştırmaktadır. Uygulama geliştirilirken güncelleme yapılması sırasında yazılım geliştiricilerin hepsine haber verilmesi gerekmektedir. Aksi durumda süreçlerin koordinasyonunda aksamalar yaşanması muhtemeldir Büyük çaplı projelerde aynı uygulama üzerinden geliştirilme yapılması farklı projelere entegrasyon konusunu da güç durumda bırakmaktadır.

2.2. Mikroservis Mimari

Mikroservis terimi 2011 yılında bir yazılım mimarları çalıştayında ortak bir mimari tarz olarak görüşüldü. Mayıs 2012 yılında aynı yazılım mimarları tarafından adına karar verildi (Lewis ve Fowler, 2014). Mikroservis mimarinin temeli dağıtık sistemlerin ortaya çıkmasına dayanmaktadır. Dağıtık sistemler ölçeklenebilirlik, esneklik ve modülerlik gibi kavramlar ile mikroservis mimarinin temelini oluşturmaktadır. Dağıtık sistemler ağa bağlı bilgisayarlar ile iletişim kurarlar ve koordinasyonu sağlarlar. Bir ağ bağlantısı ile bilgisayarlar birbirlerinden mekansal olarak ayrılmışlardır. Aynı binada veya aynı odada olabilirler (George, Jean, Tim ve Gordon, 2012). Bu sistem küçük parçalara bölünebilen bir mimari düşüncesi sunmaktadır. Dağıtık sistemlerin yaygınlaşması ile beraber Servis Odaklı Mimari (SOA) yaklaşımı ortaya çıkmıştır. Servis Odaklı Mimari, farklı uygulamalar içerisindeki dağıtık sistemlerin düzenlenmesi ve kullanılmasını sağlar. Çok yönlü geliştirme imkanı sunması ile çeşitli görevleri yerine getirebilir ve sistemde esneklik sağlar. Gevşek bağlı olmaları sebebiyle birbirlerinden etkilenmeden tekrar kullanılabilmektedir. Kullanıcılar arasında tutarlı bir deneyimleri yaşatmaktadır. Servisler sistemin işleyişi servis sağlayıcılar tarafından erişime açıktır (Ramanathan ve Korte, 2014).



Şekil 14. Servis Odaklı Mimari Şeması

Kaynak: Yazar tarafından oluşturulmuştur.

Kurumsal veri yolu (EBS) olarak adlandırılan, Servis Odaklı Mimari'nin uygulanması için altyapısal olarak hizmet eden, entegrasyonun kolayca uygulanması için kullanılan bir alt yapı mimarisidir.

SOA uygulamalara göre değişen farklı rollere sahiptir. İşletmeler için müşteriler ve tedarikçilerin kullanabileceği şekilde çözümler oluşturur. Kurumsal mimariler için, modülerlik, tekrar kullanılabilirlik, gevşek bağlantı sağlaması, şekillendirilebilmesi için ele alınır. Proje yöneticileri, iş analistleri, kalite güvence mühendisleri SOA'yı kendilerine göre yorumlamaktadır. Roller için farklı tanımlara sebep olmaktadır (Darmawan vd., 2008).

Bilgi teknolojilerinde ağ bağlantısı ile birbirlerinden farklı ortamlarda etkileşim halinde çalışarak hizmet olarak sunulması ile oluşan mimari yaklaşım olan SOA, farklı uygulamalar ile esnek, modüler ve tekrar kullanılabilirlik ilkelerine bağlı olarak çalışmaktadır. Bilgi teknolojilerinde bu mimarinin kullanımı ile iş süreçlerinde zamandan tasarruf, stratejik işlere odaklanılması, geliştiricilerin iş performansında artış, hata yönetimindeki ilerleyiş ile beraber genel verimliliğin artmasını sağlamaktadır. Büyük ve karmaşık uygulamalarda projelerin farklı teknolojiler ve sistemler içerisinde entegrasyonu

ve Bilgi Teknolojileri (IT) altyapısının bağımlılığını azaltmak için kullanılmaktadır. Teknolojinin sürekli gelişmesi ile beraber yeni koşullara adaptasyonda kolaylık sağladığı içinde sıklıkla tercih edilmektedir. Mimaride belirli bir işlevi yerine getiren servisler bağımsız olarak çalışmaktadır. Örneğin, e-ticaret uygulamasında ödemelerin tamamlandığını kontrol eden servis ile siparişin oluşturmasını sağlayan servisler bağımsız olarak çalışmaktadır. Her servis diğer servisler ile ağ üzerinden iletişim sağlamaktadır. Servislerin birbirlerine az bağımlı olması ile sadece tanımlanmış olan arayüzler üzerinden iletişime geçmektedir. Sıkı sıkıya bağlı olmayan servislerin, bakımı, geliştirilmesi ve güncellenmesi daha kolay yapılmaktadır. Dış dünyaya gösterimleri mevcut olan arayüzler ile sağlanmaktadır. İletişim için genellikle XML, JSON gibi veri formatları kullanılırken, SOAP, REST gibi protokoller aracılığıyla servisler arası iletişim gerçekleşmektedir. Servisler arasında iletişim sağlanırken iş akışlarının koordinasyon içerisinde çalışması gerekmektedir. Örneğin, e-ticaret uygulamasında sipariş verildikten sonra stok kontrolü yapılması, ödeme işleminin alınması ardından kargo firması ile iletişim sağlanması uyum içerisinde çalışılması gereken bir süreçtir. Servisler tekrar tekrar aynı iş senaryoları üzerinde kullanılacak şekilde tasarlanmıştır.

Hizmet odaklı mimari ve Mikroservis mimarisi, her ikisi de uygulamaları yönetilebilir, bağımsız hizmetlere ayırma fikri ile bir yaklaşım sunmaktadır. İki mimarinin de benzerlik ve farklılıkları bulunmasına karşın uygulamalar üzerindeki amaçları ve iş süreçleri üzerinde nasıl uygulandığını incelemek gerekmektedir. Her iki mimari de büyük ve karmaşık sistemlerin bölünerek yönetilmesi konusunda modüler bir yapıya sahiptir. Modüler olarak oluşturulan bu yapı içerisinde servisler bağımsız olarak geliştirilebilir ve dağıtılabilirler. Bu bağımsız servis yapısı ile farklı servisler paralel olarak çalışabilirler. Servislerin kullanılması için benzer arayüzler ile etkileşim sağlarlar. İki mimari'de monolitik mimari ve büyük yazılım projelerinin sistemlerindeki sorunları çözmeleri konusunda benzer faydalar sağlamaktadır. SOA daha büyük ve kapsam olarak genel hizmetleri içerirken, mikroservisler belirli bir işlev üzerine çalışan servisler inşa etmeye odaklanmışlardır, sorumlulukları daha küçük bir alanı kapsamaktadır. İletişim protokolleri de mimariler arasında farklılık göstermektedir. SOA genellikle daha kompleks protokoller ile güvenlik ve işlem yönetimi gibi konular üzerinde dururken

mikroservisler Hipermetin Transfer Protokolü (HTTP) tabanlı farklı çeşit API'ler ile daha hızlı şekilde yanıt vermesine odaklanarak iletişim sağlar. Mikroservis mimarilerde her servis kendine ait bir veritabanına sahip iken, SOA merkezi olarak veritabanı kullanır ve aynı veriler üzerinde çalışmasına imkan sağlar.

Servis Odaklı Mimari, birbirleriyle doğrudan etkileşim halinde olmayan uygulama yazılımlarında aynı verilerin formatları ile haberleşerek etkileşimde bulunurlar. Bu sayede farklı cihazlar ile yazılımların iletişimi kolay hale gelmektedir. Örnek olarak, akıllı telefon uygulamasında bulunan yazılım ile web tabanlı bir tankta bulunan yazılım uygulamasının servis tabanlı mimari ile yazılım sistemlerinin haberleşmesinin sağlanması ile bağımsızlıkları sağlanabilmektedir. Farklı platformlarda çalışan uygulamaların sıklıkla kullanılan örneği, mobil işletim sistemleri üzerinde bulunan yazılımlardır. Mobil cihazlar birbirlerinden bağımsız olarak geliştirilebilir ve haberleşme servisi servis tabanlı mimari ile geliştirilmesi durumunda bağımsız yazılımlara sahip olarak sorunsuz bir şekilde iletişim kurmaktadırlar. Bu sayede, bağımsız olarak yazılım sistemlerinin yönetilmesini sağlamış durumdadırlar (Aksu, Gündüz ve Ayanoğlu, 2013).

Mikroservis mimari prensip olarak uygulamaların küçük ve bağımsız parçalar halinde uygulanmasını ifade etmektedir. Her mikroservis, bir konuda içerik sağlar ve içeriklerin bulunduğu servisler birbirleri ile bir araya gelerek bir uygulamayı inşa etmektedir. Mikroservisler birbirine bağlı olmayan küçük parçalardan meydana gelmektedir. Ölçeklendirme, hata yönetimi, dağıtımı gibi konular parçalara ayrılarak yönetilmesi mikroservislerde kolaylık sağlamaktadır. Bağımsız olarak dağıtabilen, güncellenebilen servisler esneklik sağlar ve birbirlerini etkilemeden geliştirme yapılmasına olanak tanır. Uygulama geliştirilirken oluşturulmuş olan servislerin bağımsız şekilde çalışabilmesi sonucu elde edilen esneklik ile hızlı bir şekilde geliştirme yapılmış olan değişikliklerin güncellenmesini mümkün kılmaktadır. Mikroservisler arasındaki iletişim HTTP, RPC, RESTful API'ler aracılığıyla sağlanır. Bu protokoller servisler arasındaki iletişimi sağlar. Mikroservis mimarisinde, test aşaması, dağıtımı, güvenlik ve hata yönetimi gibi süreçler otomatik olarak gerçekleştirilebilmektedir. Geliştirme ve Operasyon olarak adlandırılan Geliştirme ve İşletme (DevOps) süreçlerin otomatik olarak

iyileştirilmesine katkı sağlamaktadır. Örnek bir senaryo da; indirim yapmış olan bir e-ticaret sitesinin kullanıcıların çok fazla giriş sağlaması ile beraber otomatik olarak sunucu üzerinde bulunan yükün ölçeklendirilmesi ve hata vermesi durumunda yeniden denenmesi gibi durumlar otomatik olarak iyileştirme kapsamında ele alınması gerekmektedir. Sistem üzerinde sürekli olarak optimizasyon sağlamalıdır. Her bir mikroservis belirli bir işlevselliği sağlamaktadır. Uygulama geliştirilirken kodlama aşamasında fonksiyonlar, sınıflar ve modüller tutarlılık ve bütünlük içinde bir servisi inşa etmektedir. Her servisin kendine özgü işlevselliğiyle, kendi içinde aynı amaca uygun hareket etmektedir. Birbirlerine tutarlılık ile bağlı olan mikroservisler tek bir sorumluluk etrafında oluşturulmuş olduğundan dolayı dışa bağımlılık minimum seviye de tutulmaktadır. Kendi içerisinde geliştirme yapıldığı için bakımı daha kolay olmaktadır. İhtiyaç dahilinde gerekli olan mikroservis sadece ölçeklendirileceğinden dolayı yönetilmesi etkin şekilde yapılabilmektedir. Mikroservisler kendi içlerinde işlevlerine göre ayrıştığından dolayı daha kolay şekilde hangi amaca uygun olarak geliştirildiği kolaylıkla anlaşılabilmektedir.

Mikroservisler birbirinden bağımsız olarak çalışabileceği için merkezi bir veritabanına sahip olma zorunluluğu bulunmamaktadır. Monolitik uygulamalarda, merkezi veritabanı ile uygulamalar geliştirilmektedir. Bu sebeple uygulama merkezi veritabanına bağımlı hale gelmektedir. Mikroservis mimari de veritabanı çeşitlilik göstermesi sebebiyle bağımsız olarak geliştirme sağlanır. Her mikroservis kendisine en uygun teknoloji seçimi ile ihtiyaçlarına uygun şekilde veritabanı seçilebilmektedir. Kendisine uygun olarak seçilen veritabanı ile etkileşim en hızlı şekilde sağlanarak performans kayıplarının da önüne geçilmektedir.

Servisler bağımsız parçalar halinde birbirlerinden ayrı şekilde oluşturulurken işlevselliklerine dikkat edilmesi gerekmektedir. Servislere gereğinden daha fazla iş yükü verilmesi ile verimlilik düşürülebilir ve sistem üzerinde karmaşıklığa yol açılabilmektedir. Birbirlerine bağımlı servis sayısının çok olması sistem üzerinde hata ayıklama sürecini uzatabilmektedir. Bu durumda güvenlik ve tutarlılık gibi kavramlar risk altında olmaktadır. Veri bütünlüğünün sağlanması için tutarlılık mekanizmalarına ihtiyaç duyabilmektedir. Bu mekanizmalar ile günlük işlem geçmişi, veri değişiklikleri sırasında

tüm servislere bilgi verilmesi, çakışma durumunun yönetilmesi gibi önemli sorunların önüne geçilmektedir. Uygulamalar kendi gereksinimlerine bağlı olarak farklı şekilde mekanizmalar tercih edebilmektedir.

2.2.1. Mikroservis Mimarinin Avantajları

- Optimizasyon ve Performans

Mikroservis mimarisinde ağ üzerindeki iletişim performans için önemli bir faktördür. Bilgisayarın hafızası tarafında yapılan işlemler, ağ tarafında yapılan işlemlerden daha hızlıdır. Fakat mikroservisler bağımsız ve daha küçük parçalar halinde servislerden oluştuğu için ağlar aracılığıyla iletişim kurması gerekmektedir. Bir sistemde, ağlar aracılığıyla ne kadar fazla iletişim kuruluyorsa o kadar çok performans azalmaktadır. Fakat, mikroservislerin ne kadar iyi derece bölündüğü önem arz etmektedir. Sınırları iyi şekilde belirlenmiş olan mikroservisler, gereksiz iletişim sağlamayarak performansı arttırır. Kendi işlevini bağımsız olarak yerine getiren servisler iletişimi azaltacağı için performans olumlu olarak etkilenmektedir (Dragoni vd., 2017).

- Dayanıklılık

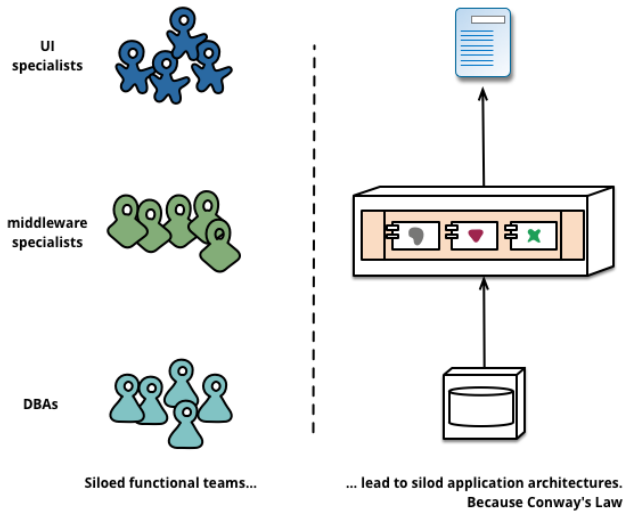
Monolitik bir mimari büyük ve karmaşık yapılı uygulamaları küçük parçalara bölerken birlikte çalışabilirlik uygulamanın işlevselliğini arttırmaktadır. Bu mimari ile dayanıklılık ve hata izolasyonu ile meydana gelen sorunun sadece yaşandığı servis içerisinde bağlı kalmasını sağlar. Örneğin, ödeme servisinde yaşanan bir sorun ile uygulamanın arayüzünde bulunan ürünler listesinin etkilenmeyerek ürünlerin gösterime devam etmesi gerekmektedir. Kullanıcı deneyiminin olumsuz sonuçlanmaması gerekmektedir. Ayrıca bağımsız olarak oluşturulan servisler ile yaşanan problemin hangi servis tarafından yaşandığının hızlı bir şekilde müdahale edilerek çözümlenmesi önem arz etmektedir. Müdahale sürelerindeki artış dayanıklılığı arttırmaktadır. Servisler üzerindeki yük miktarının uygulamanın belirli dönemlerinde artışı da kaynak artırımı ile verimli kullanılmasını sağlayıp esneklik ve ölçeklik açısından dayanıklılığın artışına destek olmaktadır. Dünya genelinde kullanılan önemli uygulamaların kesintiye uğramaması

kritik önem arz etmektedir. Ödeme sistemleri, müşterilerin hesap yönetimi, sağlık hizmetlerinde tıbbı teşhis ve randevu sistemi, hasta bilgilerinin yönetilmesi gibi konularda sistemlerin güvenli şekilde yönetilmesi gerekmektedir.

- Bağımsız Çalışabilirlik ve Yeni Teknoloji Kullanımı

Mikroservislerin bağımsızlık olan çalışabilirliği ile tek bir veritabanı ya da yazılım diline bağlı olma gerekliliğini ortadan kaldırmaktadır. Teknolojiler ihtiyaçlar doğrultusunda ortaya çıkmaktadır. Veritabanları, yazılım dilleri belirli gerekliliklere göre seçilmesi gerektiği üzere bağımsız seçimler teknoloji bakımından daha çevik bir uygulama geliştirme süreci ortaya koymaktadır. Uygulama geliştiriciler farklı servisler üzerinde ayrışarak ekip çalışması yaptığında ihtiyaçlara göre yeni teknolojilere geçilmesine de olanak sağlamaktadır.

Yöneticiler genellikle uygulamaları teknolojilere göre ayırmaktadır. Veritabanı, kullanıcı arayüzü, sunucu kısmı gibi ekiplere ayırmaktadır. Ekiplerin bölünmesi ile küçük çaplı işler bütçe gerektiren işlere dönüşebilmektedir. Yetenekli bir ekip işleri minimize ederek, kaybı en aza indirecek şekilde olan çözümü seçerek ilerlemektedir. Bu mantık Conway Yasası'nın uygulanmış halidir (Lewis ve Fowler, 2014).



Şekil 15. Conway Yasasının İşleyişi

Kaynak: Lewis, J. ve Fowler, M. (2014). *Microservices: A definition of this new architectural term.* 21 Ocak 2024 tarihinde https://martinfowler.com/articles/microservices.html adresinden edinilmiştir.

2.2.2. Mikroservis Mimarinin Dezavantajları

- Servisler Arası Güvenlik Sorunu

Uygulama kendi içerisinde farklı servislerden oluşması genel sistemin güvenliğinin azalmasına yol açabilmektedir. Her mikroservis kendi içerisinde veri alışverişinde bulunurken rutin olarak güvenlik kontrolü yapması gerekmektedir. Bazı güvenlik kontrolleri kişi bazında olacak şekilde özelleştirilmiş olarakta uygulanabilmektedir. Kendine özgü oluşturulan doğrulama sistemleri ile yetkilendirmeler koordinasyon bakımından sorun yaşayabilmektedir. Servisler arasında izolasyonun sağlanması ile ancak güvenlik açıklarının önüne geçilebilir. API'ler üzerinden iletişim gerçekleştiği için genellikle doğru şekilde korunması ve izlenilmesi önemlidir.

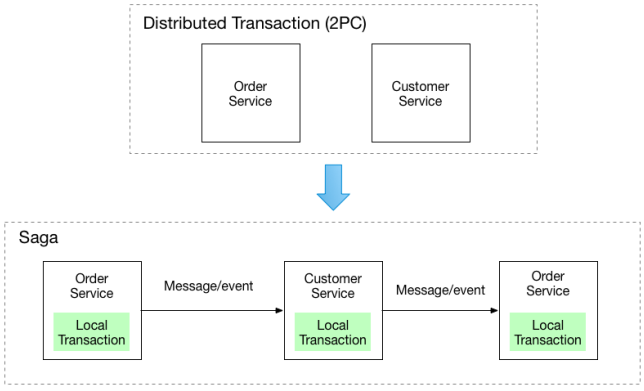
- Dağıtımda Zorluklar Yaşanması

Yazılım mimarileri güncellenebilir, bağımsız şekilde değiştirilebilir olarak tasarlanmaktadır. Uygulamalar oluşturulurken kullanılan kütüphaneler doğrudan çalıştırılacak şekilde oluşturulmuş olan fonksiyonlardır. Servisler uygulamanın içerisindeki bileşenlere bağlı olmaksızın, bu bileşenler ile uzaktan erişim yolu ile genellikle çağrıda bulunurlar. Servisler, kütüphanelerden farklı olarak bağımsız bileşenler olarak ifade edilmektedir. Servisleri bağımsız olarak kurulabilir olması da farkları arasındadır. Fakat, bir kütüphaneyi güncellemek için uygulama üzerinde tekrardan kurulum yapılması yükünü ortaya çıkarmaktadır. Her koşulda tekrardan kurulum yapılmasına olanak olmayabilmektedir. Servislerin arayüzünün değişmesi durumunda, yeni oluşturulmuş olan servisin güncellemesini yapmaktan daha çok çaba gerektirmektedir. Mikroservis mimarisi için arayüz değişikliğinin yapılmasının yarattığı etkileri azaltmak için uygun şekilde tasarlanması gerekmektedir (Dmitry, Manfred, 2014). Servislerin arayüzlerindeki değişim domino etkisiyle güncellemeye ihtiyaç duyar. Bu ihtiyaç doğrultusunda dağıtım süreçleri karmaşık bir hal almaktadır. Büyük çaplı uygulamalarda dağıtım ve genel entegrasyonların sağlanması koordinasyonu zor bir hale getirmektedir.

- Veriler Arasındaki Tutarsızlık

Veritabanları her mikroservisin kendi içerisinde ayrı olacak şekilde bulunmaktadır. İlişkisel veritabanlarındaki operasyonlar için Transaction yönetimi önemli bir konudur. Bütünlük, tutarlılık, bağımsızlık, dayanıklılık (ACID) veritabanlarındaki verilerin bütünlüğünü temsil etmektedir. Veritabanı işlemlerinde hatasız şekilde gerçekleşmesini ifade eden prensip için her harf bir durumu temsil etmektedir. Veritabanı yönetim sistemlerinde sağlık, ödeme sistemleri, finans gibi konularda veri tutarlılığı kritik önem arz etmektedir.

Verilerin hatalarının önüne geçilmesi ve tutarsızlık yaşanmaması için gerçekleşen işlemlerin atomik yapıda olması gerekmektedir. Bir işlem başladığında ya işlemin tamamen tamamlanması ya da hata durumunda işlemin gerçekleştirilmemesi gerekmektedir. Monolitik mimari de ACID işlemleri kolaylıkla uygulanabilmektedir. Mikroservis mimarinin en kritik dezavantajı dağıtık çalışan servislerin verileri arasında nihai tutarlılığın sağlanması gerekliliğidir. Birden fazla servisin güncellenmesi için oluşturulmuş olan komutun uygulanması gerekmektedir. Veritabanlarında veri tutarlılığının sağlanmasının dışında her servisin veritabanındaki çeşitli sorgularda yaşanan problemlerinde öncesinde çözümlenmesi gerekmektedir. Örneğin, e-ticaret uygulamasında kredi limiti kontrolü sağlanırken müşterinin sipariş aşamasında kredi limiti kontrol edilmelidir. Bu iki aşama farklı veritabanlarında bulunduğundan dolayı kontrolü sağlanmalıdır (Richardson, 2017) .



Şekil 16. Hizmete Özgü Veritabanı Şeması

Kaynak: Richardson. (2017). *Pattern: Microservice architecture.* 5 Kasım 2023 tarihinde https://microservices.io/patterns/microservices.html adresinden edinilmiştir.

2.2.3. Mikroservis Mimarisinde İletişim

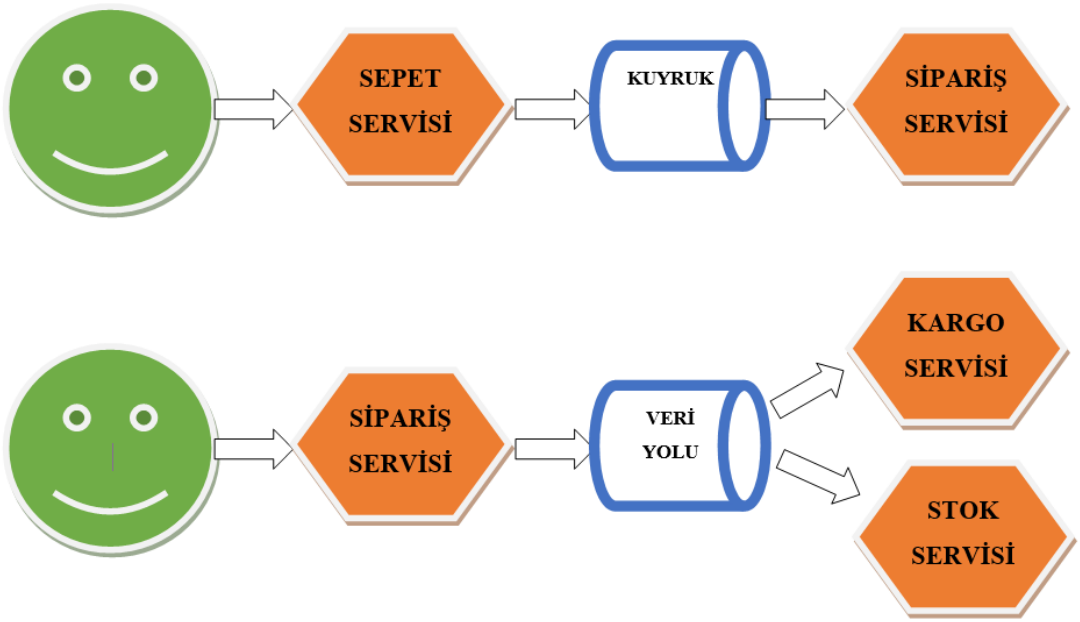
Mikroservis mimarisi, büyük ve karmaşık yapılı uygulamaları daha esnek, ölçeklenebilir, yönetilmesi daha kolay olacak şekilde modern teknolojilere uygun olarak bölünmesine yardımcı olmaktadır. Servislerin kendi içerisinde yaşam döngüsüne sahip olması sebebiyle servisler arası iletişimin iyi şekilde yönetilmesi önem arz etmektedir. Servisler arasında iletişim sistemin güvenliği, performansı ve dayanıklılığı gibi konularda etkili iletişim sağlanması gerekmektedir. Doğru iletişim yöntemlerinin seçilmesi ile teknolojilerilerin gelişmeye devam etmesi ile düzenli olarak iyileştirilmesi sağlanmalıdır. Servisler arası iletişim, uygulamaların sürekliliğini sağlamak açısından önemlidir. HTTP/HTTPS protokolü ile RESTful API'ler, Uzak Prosedür Çağrısı (gRPC) ya da asenkron sistemler gibi servisler arası iletişim sağlanır.

2.2.3.1. Asenkron İletişim

Mikroservis mimarisinde, servisler arasında iletişim kurulurken senkron ve asenkron olarak iletişim gerçekleştirilmektedir. Asenkron iletişim, mesajı gönderen kişinin mesajı ilettikten sonra geriye bir dönüş beklemediği iletişim şeklidir (Van Steen ve Tanenbaum, 2017). Asenkron iletişim çift yönlü olarak gerçekleşiyor ise, örneğin bir müşterinin isteği gönderilir ve işlem uzun sürecek şekilde ise bu istek sunucu tarafından gerçekleştirilirken müşteri yanıtı beklemek yerine, başka işlemlerle de meşgul olabilmektedir (Cebeci, 2019). Asenkron iletişimde servisler geri dönüş beklememektedir ve zaman farkına bakılmaksızın veri alışverişi sağlamaktadır. Asenkron iletişim kullanılan teknolojilere göre değişiklik göstermektedir. İletişim sırasında hata olması durumunda sistem kesintiye uğramamaktadır. Servisler birbirlerinden haber beklemedikleri için bağımlılıkları bulunmamaktadır, mikroservis mimari için bu durum önemli bir avantaj sağlamaktadır. Servisler sürekli olarak proaktif durumda olmaya ihtiyaç duymamaktadır. Servise gönderilen verinin büyüklüğünden bağımsız olarak aşırı yüklenmeye bağlı olarak istekler düşmemektedir. Uzun sürecek olan işlemler arka planda gönderilmeye devam edilmektedir. Bu sebeple, hatalara karşı daha dayanıklıdır. Kuyruk yapısı sayesinde mesajlar saklanabilir ve hata giderildiğinde mesajın gönderimi sağlanabilmektedir.

Asenkron iletişimde, sunucular üzerinde kaynakların ekonomik olarak harcanmasını ve kullanımın yüksek olduğu zamanlarda değil, kullanımın az olduğu zamanlarda gerçekleştirilerek kullanılmasıyla optimizasyon sağlanmış olur.

Asenkron iletişim bire bir ve bire çok olarak iletişim gerçekleştirilmesine olanak sağlar.



Şekil 17. Tek Alıcı ve Çoklu Alıcıya Dayalı Asenkron İletişim Şeması

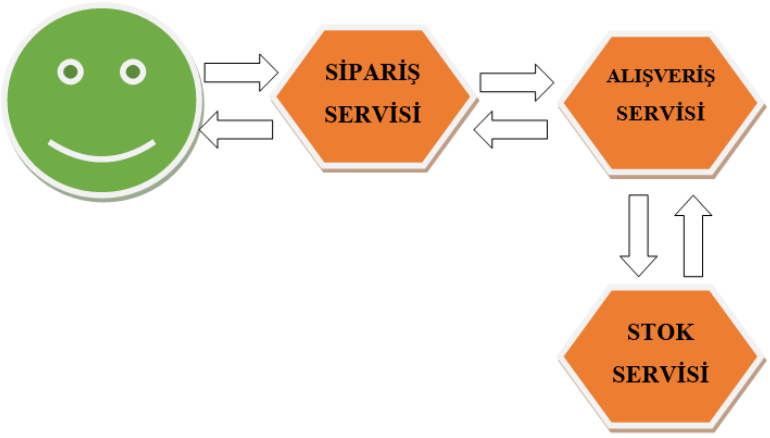
Kaynak: Yazar tarafından oluşturulmuştur.

Asenkron iletişimde kullanıcı alıcıya bir bildirim gönderdiğinde olay yayınlar. Örneğin ürünün sipariş edilmesi bir olayken, sipariş edilecek olan ürünün kargoya verilmesi ve stoktan düşülmesi eş zamanlı olarak gerçekleştirilebilmektedir. Rabbitmq, Kafka gibi sistemler asenkron mesajlaşma teknolojisi olarak tercih edilmektedir.

2.2.3.2. Senkron İletişim

Eşzamanlı iletişim mikroservis mimarisinde çok sık tercih edilmektedir. Genel olarak istek/yanıt etkileşimi olarakta tanımlanmaktadır. Bu iletişim türünde bir servis başka bir servise istekte bulunmaktadır. İstekte bulunduğu servisten geri dönüş alana kadar beklemektedir (Shafabakhsh, Lagerström, Hacks, 2020). Monolitik uygulamalarda

istekler tek bir yapı üzerinden istek sağladığı için tek bir parça olarak geri dönüş sağlamaktadır. Fakat mikroservis mimarisi birden fazla birbirlerinden bağımsız olarak farklı işlevleri yürüten servisten oluşmaktadır. Eşzamanlı iletişim birbirlerinden bağımsız olarak işlem yürüten servislerde iletişimi sağlamak için sıklıkla kullanılan yöntemlerden biridir. Servisler birbirlerinden haber bekledikleri için dönüş sürelerinde yaşanan gecikmeler sonucunda başarısız dönüşü alınabilmektedir. Sistem üzerinde ciddi boyutlara ulaşabilecek olan gecikmeler maliyet ve kullanıcı kaybı olarak yansımaktadır. Ardarda yapılması gerekli olan işlemlerde birbirlerini bekleme durumu yaşandığı için sistemsel gecikmeler yaşanabilmektedir.



Şekil 18. Senkron İletişim Şeması

Kaynak: Yazar tarafından oluşturulmuştur.

Eşzamanlı iletişim geri dönüş değeri beklediğinden dolayı sistemde bloklamaya ve hata oluşumuna sebep olabilmektedir. Bu sebeple gerçekten gerekli olduğu durumlar dışında sistem içerisinde iletişim türü seçilirken dikkatli olunması gerekmektedir. Örneğin, bir e-ticaret uygulamasında mikroservis mimarisinde senkron iletişim çok sık tercih edilmektedir. Müşterilere hızlı geri dönüş sağlaması önem arz etmektedir. Ürün sepete eklendikten sonra güncel fiyat, stok durumu hemen kontrol edilmelidir. Sipariş verilmesinin ardından ödeme işleminde bankalardan geri dönüş alınıp hemen işleme alınmalıdır. Hatalı işlemlerin olması durumunda düzeltmeler, tekrar denemelerin yapılması gerekmektedir. Müşterilerin satın aldıkları ürünler için tekrar iletişime geçmesi

durumunda servisler ve iş süreçleri hakkında bilgi edinilmesinde hızlı ve kullanıcıların deneyimini iyileştirmek adına senkron mimari tercih sebebi olmaktadır.

2.2.4. Mikroservis Mimarisinde İletişim Protokolleri

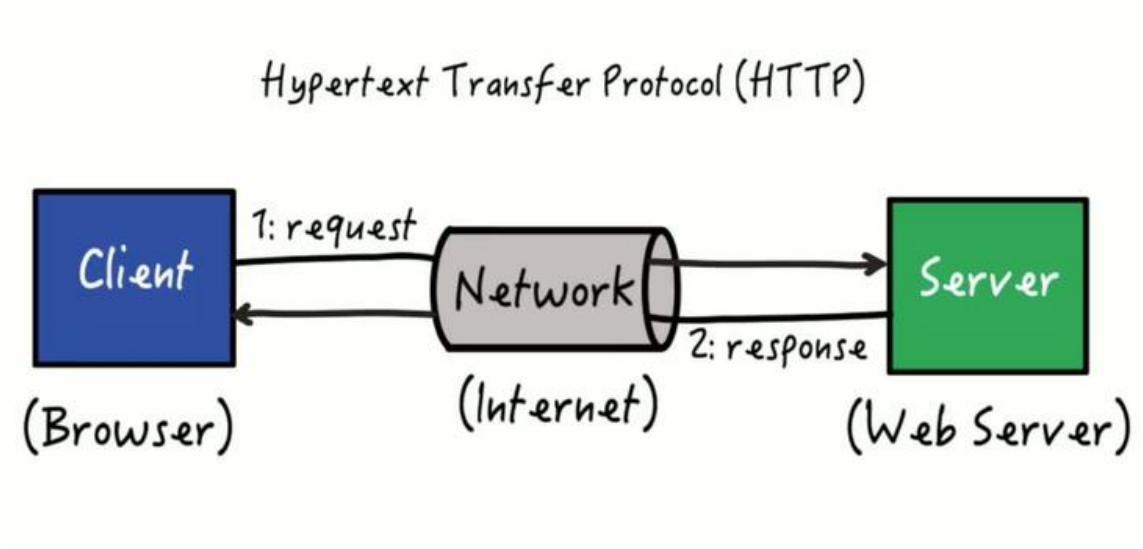
Mikroservis mimarisinde küçük parçalar halinde servislerin yönetilmesi gerekmektedir. Fakat yönetilecek olan servisler birbirlerinden ayrılmaları ile iletişim halinde bulunmaları yeni teknolojiler ile beraber gün geçtikçe üzerine çalışılan ve geliştirilmeleri devam eden bir problemdir. Bir dizi zorluk servislerin bölünmesi ile meydana gelirken farklı servisler arasındaki iletişimde nasıl etkileşim sağlayacağı gibi sistemsel değişikliklerin yönetimi ve paylaşımın ne şekilde yapılacağı da önemli bir konudur. Mikroservis mimarisinin temel yapı taşlarından biri iletişim protokolleridir ve uygulamanın ihtiyaçları doğrusunda doğru protokolü seçmek önemlidir. Uygulama ile beraber uyumlu şekilde çalışması gerekmektedir.

2.2.4.1. HTTP Protokolü

Hipermetin Transfer Protokolü (HTTP) kullanılacak olan kaynak üzerinden paylaşımı sağlanan ve birden fazla alanda kullanıma uygun şekilde, bilgi teknolojileri içerisinde uygulamalarda kullanılacak olan iletişim protokolüdür (Fielding vd., 1999). HTTP kullanımında resimler, metin dosyaları gibi içeriklerin istemci ve sunucu arasında gönderilen istekler ile iletimini sağlar.

HTTP protokolü, World Wide Web'in ortaya çıkışıyla insanlar tarafından yaygınlaşmasıyla ortaya çıkmıştır. Tim Berness-Lee 1989 yılında CERN'de HTTP üzerine çalışırken bir yandan da Hiper Metin İşaretleme Dili (HTML) için geliştirmeler ile web teknolojileri alanında çalışmaktadır. İlk olarak HTTP/0.9 ardından HTTP/1.0 ve HTTP/1.1 olarak yeni versiyonların geliştirmelerine devam etmektedir. HTTP/2.0 ile geliştirmeler devam ettikçe performans sorunlar azalmaya başlamış ve asenkron olarak istek geliştirmeye de uygun hale gelmiştir (Natran, 2023).

Bir tarayıcı sunucuya istek atmak istediğinde aralarında bir bağlantı oluşmaktadır. Oluşan bu bağlantı ile sunucunun bu bağlantıyı kabul etmesinin ardından istemci HTTP isteği üzerinden verilerini göndermektedir. Bu verilerin gönderiminde daha fazla kaynak bulunması için yeni bağlantılar yapılmasına ihtiyaç duyulmaktadır. HTTP isteği üzerinden gönderilen bu bağlantılar ile sunucu tarafından geri dönüş sağlandığında açık olan bu bağlantı kapatılır. HTTP kendi içerisinde sunucuya gelen isteğin kim tarafından yapıldığını anlayabilecek bir mekanizmaya sahip değildir. Bu kapsamda kimlik doğrulama, çerezler ve oturumların yönetilmesi için sunucu içerisinde oluşturulmuş olan mekanizmalar kullanılmaktadır.

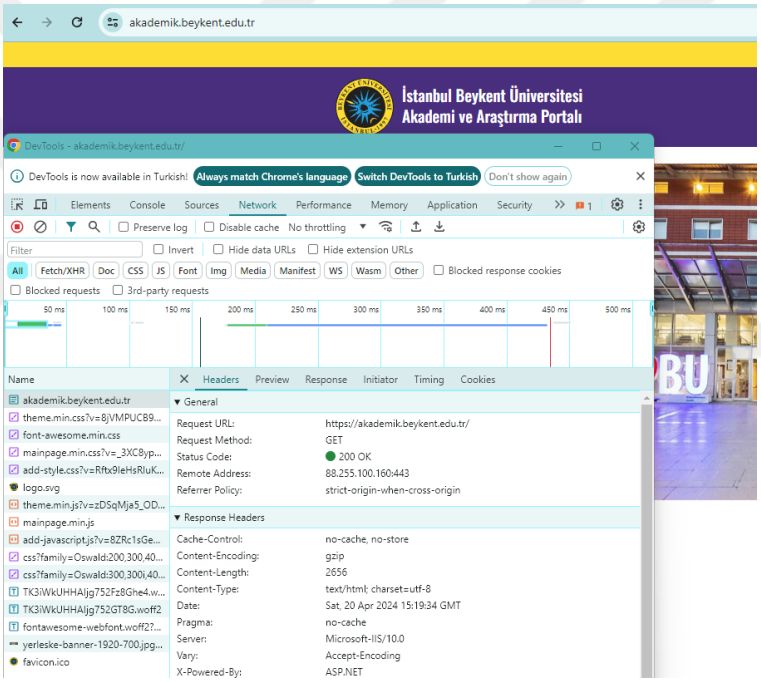


Şekil 19. Kullanıcı ve Sunucu Arasındaki İstek-Yanıt Transferi

Kaynak: Pandurang. (2022). *How does HTTP protocol work.* 20 Ocak 2024 tarihinde https://pandurangpatil.medium.com/how-does-http-protocol-work-426b1a4158f3 adresinden edinilmiştir.

HTTP üzerinden gönderilen isteklerin içerisinde isteğin bulunduğu satır Tekdüzen Kaynak Bulucu (URL) ile başlamaktadır. HTTP istekleri gerekliliklerine göre farklı yöntemler ile sağlanmaktadır. Bu yöntemlerin en sık kullanılanları GET, POST, PUT ve DELETE olarak adlandırılmaktadır. GET isteği ile sunucu üzerinde bulunan verilerin alınması istenilmektedir. POST isteği ile sunucu üzerine veri gönderilmesi istenmektedir. PUT isteği ile güncelleme yapılmış olan veri üzerindeki değişikliklerin sağlanılması için kullanılır. DELETE isteği ile ile bir kaynaktaki verilerin geri dönüştürülemez biçimde

kaldırılması için kullanılmaktadır. PATCH, HEAD, OPTIONS, TRACE gibi farklı işlemleri de gerçekleştirmek için kullanılan yöntemlerde bulunmaktadır. URL'in yolunun son kısmına gönderilmek istenilen ek bilgiler için parametreler eklenmektedir. Örneğin, "http://akademik.beykent.edu.tr/talat-firlar" şekildedir. Gönderilen isteklerde isteğin kimi tarafından gönderildiği ve isteğin içerisinde bulunanlar hakkında başlık bulunmaktadır. Gönderilen isteğin içeriğinde ise isteğin gövdesindeki veriler bulunmaktadır. İsteğin verilerinin iletimi çoğunlukla gövde aracılığıyla sağlanmaktadır. Bu verilerin gönderilmesinin ardından sunucu isteği kullanarak geriye bir yanıt dönmektedir.



Şekil 20. HTTP İsteği Örneği ve İsteğin Detayı

Kaynak: Yazar tarafından oluşturulmuştur.

HTTP, Aktarım Kontrol Protokolü (TCP) protokolü ile çalışmaktadır. Bu protokolün kullanımı ile birlikte istemcinin yapmış olduğu HTTP isteği web sayfasına gönderilir ve sunucu yapılmış olan isteği alır, yapılan isteğe uygun olarak geri dönüş yanıtı gönderir. Gönderilmiş olan içeriğin birden fazla parçasını bulunduran web sayfası, istemci üzerinden bu sonucu almak için istek gönderir (Wei, Hong ve Shi, 2016). İstemci

üzerinden sunucuya gönderilmiş olan isteklerin başarılı bir şekilde geriye yanıt vermesi beklenmektedir. Geriye dönen yanıtın daha anlaşılır olabilmesi için standart olarak belirlenmiş durum kodları bulunmaktadır.

HTTP durum kodları uygulama üzerinde yapılan isteklerin anlaşılabilmesi adına 3 basamaklı olarak bulunan sayılardır. Alınan olumlu ya da olumsuz hataların sonucunda durumun açıklamasının anlaşılmasını sağlamaktadır. Durum kodlarının ilk basamağı geriye dönen cevabın sınıfını tanımlamaktadır.

1 ile başlayan kodlar, bilgilendirme amacındadır. 2 ile başlayan kodlar başarıyla işleme alındığını ifade eder. 3 ile başlayan kodlar yönlendirilme olduğunu belirtmektedir. 4 ile başlayan kodlar istemci üzerinde bir hata meydana geldiğini anlaşılmadığını ifade etmektedir. 5 ile başlayan kodlar sunucu üzerinde geçerli bulunan bir isteğin gerçekleştirilemediğini ifade etmektedir.

Tablo 1. HTTP Durum Kodları Tablosu

1xx Durum Kodları	
100	Gönderilen isteğin alındığı ve sunucu tarafından işlenmeye başladığını ifade etmektedir.
101	Sunucu istemcinin kullandığı protokolü değiştirmeyi kabul ettiğini ifade etmektedir.
2xx Durum Kodları	
200	İstemci tarafından alınan isteğin başarılı bir şekilde işlendiğini ifade etmektedir.
201	İstek sonucunun başarılı alındığını ve yeni kaynak oluşumu sağlandığını ifade etmektedir.
202	Sunucunun isteği aldığı fakat henüz işlemi bitiremediğini ifade etmektedir.
203	Sunucuya doğrudan erişim sağlanamadığı ara sunucu tarafından isteğin alındığını ifade etmektedir.
204	Sunucu tarafından işlemin alındığını fakat geriye içerik döndürmediğini ifade etmektedir.
205	Sunucu tarafından işlemin alındığını fakat sayfanın yeniden yüklenmesi gerektiğini ifade etmektedir.
206	Sunucunun büyük bir veriyi parça halinde işleme alındığını ifade eder. Başarılı sonuç döndürür.

207	Sunucu üzerinde bir kaynaktan birden fazla işlem durumuyla yapılan işlemlerin sonucunu ifade etmektedir.
3xx Durum Kodları	
301	İstemci üzerinde talepte bulunulan kaynağın başka bir URI'de olduğunu yeni URI'ye yönlendirilmesi gerektiğini ifade etmektedir.
307	İstemci tarafından sunucuya gönderilen data üzerinde yönlendirme yapılmak istendiği için kullanılır. Farklı bir URI'ye yönlendirme yapılmıştır.
308	Kalıcı olarak gelen isteğin başka bir URI'ye yönlendirme yaptığını ifade etmektedir.
4xx Durum Kodları	
400	Sunucu gelen isteği anlamadığını, isteğin hatalı olduğunu ifade etmektedir.
403	Sunucu gelen isteği gerçekleştirmek için izne sahip olmadığını ifade etmektedir.
404	İstenilen kaynağın sunucu tarafında bulunmadığını ifade etmektedir.
5xx Durum Kodları	
500	Sunucunun isteğin işlendiği esnada hata aldığını ve işlem yapamadığını ifade etmektedir.
501	Gelen isteği işleyebilecek kadar sunucunun yeteneğinin olmadığını ifade etmektedir.
502	Sunucu gelen isteği başka bir sunucu üzerinde işlemeye çalışırken hata aldığını ifade etmektedir.
504	Sunucunun başka bir sunucuya erişimi üzerinde zaman aşımına uğradığını ifade etmektedir.
505	Sunucu tarafında istemcinin HTTP protokolünün kullanmış olduğu versiyonun desteklenmediğini ifade etmektedir.
511	Sunucunun istemci tarafından gelen isteği işlemeden önce kimlik doğrulamasına ihtiyaç duyduğunu ifade etmektedir.

Kaynak: Yazar tarafından oluşturulmuştur.

2.2.4.1.1. HTTPS Protokolü

HTTP durum kodlarının güvenli sürümü olarak Güvenli Hipermetin Transfer Protokolü (HTTPS) sonuna almış olduğu 'S' harfini güvenli olduğunu ifade etmek için kullanmaktadır. Tarayıcı ve web sitesi arasında oluşturulan bağlantının şifre ile korunduğunu ifade etmektedir. Çevrimiçi alışveriş, sipariş gibi ciddi derecede gizlilik arz eden alanlarda kullanılmaktadır. HTTPS kullanıcıların iletişimini şifreli bir biçimde sağlamak için Güvenli Yuva Katmanı (SSL) gibi güvenlik protokollerini içermektedir. Şifreleme yöntemleri ile güçlü şekilde korunmaktadır. Web siteleri SSL sertifikası kullanarak güvenli bir şekilde kullanıcıların oturumlarına giriş yapmasını sağlamaktadır. HTTPS'in kullanılması ile örneğin bir e-ticaret sitesinden alışveriş yapıyorsunuz ve ödeme aşamasında kredi kartı bilgilerinin korunmasını sağlar. Hassas bilgilerin güvenliği konusunda kullanıcılara memnuniyet veren deneyimler sunmaktadır.

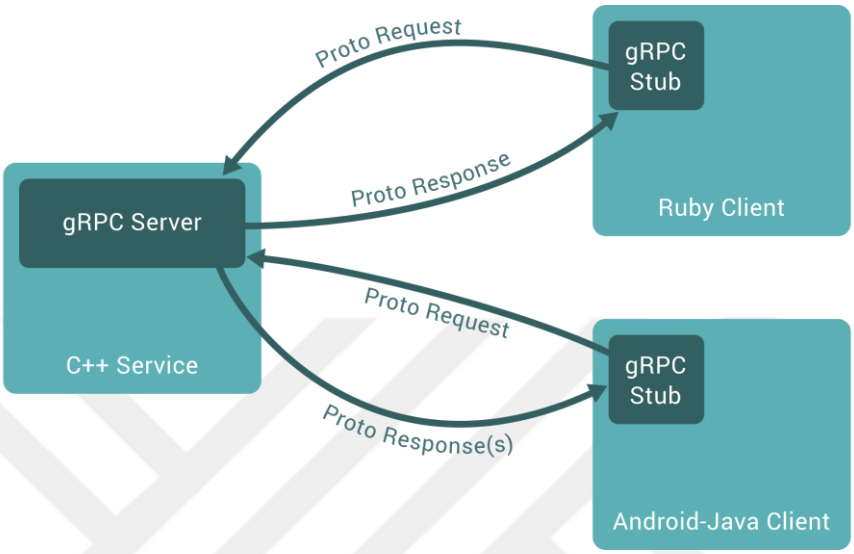
Günden güne globalleşen kitleler üzerinde kişisel verilerin önem kazanması ile birlikte internet çeşiti farketmeksizin web sitelerinin hepsinde HTTPS kullanımı artmaya başlamıştır. Gün geçtikçe önemi artarak devam etmektedir (Konigsburg, Pant, Kvochko, 2014).

Yapılan araştırmaya göre HTTP ve HTTPS'in performansları karşılaştırılmıştır ve güvenli web sunucularının daha iyi performans gösterdiği sonucu elde edilmiştir. Veri transferi sırasında ise benzer hız oranları yakalanmıştır (Goldberg, Buff, Schmitt, 1998).

2.2.4.2. gRPC

Mikroservis mimarisinde gRPC farklı yerde bulunan sunucu ya da servisin kendi servisimiz tarafından kullanılan bir metoduymuş gibi client-server arasında kullanılan iletişimin daha hızlı ve kolayca sağlanmasına olanak tanıyan bir frameworktür. RPC yeni olan bir framework değildir. Google desteği ve HTTP 2.0 güncellemesi ile açık kaynaklı ve ücretsiz olarak sunulmaktadır. Kullanılan verilerin serialization işlemi binary formatında yapılmaktadır ve Protocol Buffer olarak adlandırılan bir protokol olarak tanımlanmaktadır. Platform ve dile bağlı olmadan entegrasyon yapılmasına olanak sağlamaktadır. Kullanıcı ve sunucu arasındaki yanıt ve istekleri çift yönlü olacak şekilde

iletişimi sağlamaktadır. Dağıtık mikroservis yapılarının kullanımında hızlı ve etkili bir iletişim sağlamaktadır.



Şekil 21. Grpc Veri İletişimi Grafiği

Kaynak: gRPC Authors. (2024). *Introduction to gRPC*. 5 Şubat 2024 tarihinde https://grpc.io/docs/what-is grpc/introduction/ adresinden edinilmiştir.

Mikroservis mimarisinde gRPC dağıtık uygulamaların içerisinde basit ve anlaşılır şekilde oluşturulmuş ara işlem mekanizması olduğu için sıklıkla kullanılmaktadır. Dağıtılmış uygulama mimarisi hazırlayan tasarımcılar arasında da kabul görmüş ve kullanılmaktadır. RPC mekanizması genel olarak kullanılan mekanizmalardan farklı olarak isteği gönderen ve isteği alan farklı bilgisayar veya aynı bilgisayar olabilmektedir. Gelen ya da giden isteklerin sonuçlarında alınan geri dönüşlerde iletişim sağlanır. Hata durumunda RPC başarısız olmaktadır fakat bir isteğin başarısız olması mevcutta yapılmış olan isteklerin başarısını etkilememektedir. Bir işlem başlatıldığı sırada yaşanan herhangi bir hata durumunda başarısız olan istek, istek yapan tarafından bilinmeksizin sürekli olarak isteğe devam edebilmektedir. Bu durum güvenilirlik konusunda şüphe yaşatmaktadır. İsteği yapan ve isteği gönderen kişi aynı adres ağında bulunmadıkları için paralel olarak çalışmalarında da sıkıntılar yaşanabilmektedir. RPC mekanizmasında paralel olarak çalışma konularındaki eksikliğin geliştirilmesi gerekmektedir. İşletim sistemlerinde dağıtık mimari de kullanılması ile birlikte sistemdeki hata toleransı artışıyla

birlikte sistemler daha güvenli hale gelmektedir. Dağıtık sistemlerde kullanılan dağıtık varlıklar olarak adlandırdığımız iletişim için gRPC kullanılmaktadır (Wang, Zhao, Zhu, 1993).

2.2.4.3. RESTful API

REST mimarisine uygun şekilde tasarlanmış olan API, prensiplere uygun olduğunda "RESTful" olarak isimlendirilmektedir. HTTP protokolü üzerinde çalışmaktadır. Veri alışverişini sağlamak için JSON yada XML gibi format çeşitlerini kullanmaktadır. API'ler birbirlerinden farklı URI'ler ile kullanılmaktadır. HTTP üzerinde uygulanan bütün metotları kullanma hakkına sahiptirler. Sunucu yapılan bütün istekleri birbirlerine bağlantısı olmadan inceler. API'ler genel olarak web uygulamalarında veriler arasında akışı sağlamak için tercih edilir. Web servisler API'ler ile kullanım kolaylığı sağlar ve daha hızlı şekilde oluşturulmasını katkı sağlamaktadır.

RESTful servisler uygulamaların sadece kendiniz tarafından kullanılıyor olmasına rağmen standart olarak kullanılması fayda sağlamaktadır. Geniş kitlelerin uygulama geliştirirken tercih ettiği standart olarak kabul görmektedir. Geliştiricilerin entegrasyon kolaylığı ile birlikte kolay anlaşılmasına fayda sağlamaktadır. RESTful olarak web servisin işlemlerin tutarlı bir şekilde olmasına fayda sağlar. Benzersiz URI'ler ile kendine özgü tanımlayıcı ve kaynaklara erişim sağlanabilmektedir. REST, kendi içerisinde önceden yapılmış olan isteklerin tutulmasını istememekte ve bu sayede güvenlik, ölçeklenebilirlik gibi konularda avantaj sağlamaktadır. RESTful servisler, Ajax istemcilerini kullanarak uygulamaların geliştirilmesine kolaylık sağlamaktadır (Richardson, Ruby, 2007).

Hizmet mimarileri için REST giderek daha fazla kullanılır hale gelmektedir. HTTP aracılığıyla veri iletişiminde ciddi kolaylıklar sağlamaktadır. REST, küçük verilerin aktarımı sırasında problem yaşatmamaktadır. Büyük verilerin aktarımında çoğunlukla darboğaza takılma sorunları yaşanmaktadır. Büyük verilerin aktarılması sırasında yaşanan yavaşlık sebebiyle kullanıcılar üzerinde memnuniyetsizliğe yol açabilmektedir. TCP protokolü ile yüksek veri bütünlüğü ve verilerin tam ve eksiksiz olarak iletilmesi

beklenmektedir. Kontrollerin sağlanması ve her veri için tek tek kontrol edilmesi RESTful servisler için yavaşlığa sebep olmaktadır (Ko vd., 2012).

RESTful servisler HTTP metodlarını kullanarak kullanıcılara kolay bir şekilde öğrenme sağlarlar. İstemci ve sunucu arasında birbirlerinden haberdar olmadan geliştirilebilmesine olanak sağlamaktadır. Bu sayede elde edilen esneklik ile uygulamaların hızla ölçeklendirilmesine imkan sağlamaktadır. Platform bağımsız olarak çalışması ile birlikte mobil ya da web ortamında sorun yaşanmamaktadır. HTTP'nin kendisine özgü özelliklerini kullanarak paralel istekleri kolaylıkla yürütebilmektedir. Servisler benzersiz alan adlarına sahip olmaları sebebiyle kullanıcılar tarafından kolaylıkla tanımlanmakta ve erişilmektedir. Mevcut mimari desenleri kullanması sebebiyle uygulamalar modüler olarak yeniden kullanılabilir hale gelmektedir. RESTful servislerin her alanda kullanımı önerilmemektedir. Örneğin, akıllı şehir teknolojileri üzerinde geliştirme yapan bir firmada sensörler, kameralar, park yeri durumları, araçların gün içerisindeki hız verileri gibi durumlar büyük bir veri seti oluşturmaktadır. Bu kapsamda veri akışını sağlamak ve iletişimi gerçekleştirmek için yeterli değildir. Mevcut ihtiyaçlar üzerinden doğru mimari ve iletişim protokolünün seçilmelidir.

2.2.5. Mikroservis Mimarisinde Hata Yönetimi

Mikroservis mimarisinde hata yönetimi, sistem üzerinde meydana gelen hataların tümünün yönetilmesi, hataların tespit edilmesi ve hataların giderilmesi süreçlerini kapsamaktadır. Mikroservis mimarisi genel hatlarıyla büyük projelerin küçük parçalar halinde yönetilmesinin verdiği avantajları kullanmaktadır. Bu avantajların kullanımı sırasında küçük hizmetlerin yönetilmesi ile ilgili olarak yaşanan hatalarında yönetilmesi konusu önem arz etmektedir. Mikroservislerin çalışmasıyla birlikte olası hataların tespit edilmesi, oluşan hataların sistem üzerinde kaydının tutulması, hata sonucu meydana gelen durumun izlenebilmesi ve takibinin yapılması gerekmektedir. Hatalar uygun durumlarda işlenir ve sistem üzerinde sorumlu olan kişilere hata mesajları ile beraber bilgilendirme yapılabilmektedir. Meydana gelen bütün hatalar geri döndürülemez vaziyette değildir.

Hatalarda kendi içerisinde programsal hatalar, veri hataları, kullanıcı hataları, altyapı hataları ve işlemsel hatalar gibi farklı kategorilere ayrılmaktadır. Servislerin birbirleri üzerinde meydana gelen hatalara karşı hata tolerans stratejileri uygulanabilmektedir. Servislerin birbirleri üzerinde sistemin devamının sağlaması için başarısız olan servisten etkilenmeden çalışmaya devam etmesi sağlanmaktadır. Otomatik yeniden yükleme, tekrar deneme gibi bir çok teknikler kullanılmaktadır. Bir serviste yaşanan hatanın başka bir servis tarafından devralınabilmesi söz konusudur. Bu tarz kritik önem arz eden hata toleranslarına ihtiyaç vardır. Örneğin, bir e-ticaret sitesinde alışveriş yaparken sipariş oluşturulması için ödemenin yapılmasının ardından kargo firmasına haber verilmesi gerekmektedir. Mevcut kargo firmasının o esnada servisinin kullanılamıyor olması durumunda alternatif olarak başka kargo firmalarının seçiminin yapılmasını sağlaması gerekmektedir. Arka planda yaşanan hatalar dışında kullanıcı tarafından da yapabilecek olan hataları göz önünde bulundurmak ve bu hatalara karşı önlem almak gerekmektedir. Sistem tarafından hatalı girişler yapılması durumunda kullanıcılara hata düzeltmesi için yönlendirmeler ile hata mesajları gösterilmektedir. Sistemde yaşanan hatalar güvenlik ihlallerine sebep olabilmektedir. Uygulamalarda güvenlik seviyesinin azaldığı durumlarda, hata toleransı stratejileri devreye alınarak sistemin açığının en kısa sürede toparlanmasına yardımcı olur.

Mikroservis mimaride servislerin küçük parçalar halinde ayrı ayrı olmasıyla hata düzeltme ve güncelleme yapılması daha kolay olmaktadır. Servislerde yaşanan hatalarda sorun olduğunda geri alınabilir ya da yeni versiyona güncelleyebilirsiniz, tekrardan uygulamayı dağıtmayı gerektirmez. Geleneksel yapıdaki uygulamalarda, hatalar mekanizmayı etkilemektedir. Bir mikroservise erişilmediğinde, çalışan diğer mikroservisleri etkilememektedir. Asıl önemli olan konu, bir mikroservisin geliştirilirken hataları ele alabilecek ve yönetebilecek şekilde geliştirilmesidir (Microsoft, 2024).

Monolitik mimari de hata yönetimi, mikroservis mimariye göre kolaydır. Monolitik uygulamalar büyük bir yapıda ve kendi içerisinde parçalar halinde olmadığından dolayı sistem üzerinde yaşanan hata genel işleyişi de etkilemektedir. Hataların takip edilmesi, loglanması, yaşanması muhtemel olan hatalar için hata

mesajlarının oluşturulması sorunların çözümünün kolaylaştırılması adına önemlidir. Hata senaryolarının testlerinin yapılması hataların azalmasına katkı sağlamaktadır. Sistem üzerinde düzenli olarak yedeklerin alınmasıyla veri kaybının önüne geçilmektedir. Uygulamanın ihtiyaçları doğrultusunda hata izleme ve analiz araçları tercih edilmesi gerekmektedir.

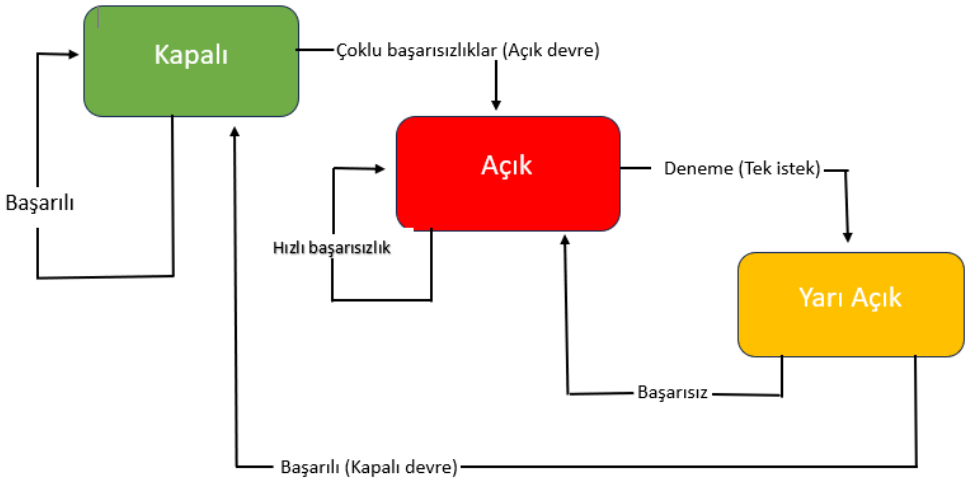
Mikroservis mimari'de hataları başarı bir şekilde yönetmek için hata yaşanabilecek durumları basit ve tekrar ele alınabilecek şekilde yönetilmesi gerekmektedir. Hatalar; geçici hatalar ve kalıcı hatalar olarak iki kısımda incelenmektedir. Geçici hatalar, sistem üzerinde yaşanan ağ sorunları, sistemsel kesintiler ve veri tabanı bağlantı sorunları gibi hatalardır. Geçici hatalar kalıcı hasarlara yol açmamaktadır. Bu şekilde yaşanan hatalar yedek sisteme geçiş sağlanması, hizmet dışı olduğunun bilgisinin verilmesi ya da yeniden sistemin başlatılması gibi çözümler sağlanmaktadır. Kalıcı hatalar ise yazılım hataları, donanım arızası, güvenlik açığına yol açan hataları ya da veri kaybının yaşanması durumudur. Bu gibi hatalara acil olarak müdahale edilmektedir. Hataların sistem üzerinde yarattığı etkiye göre müdahale süresi sebebiyle sistemin normal işleyişine dönmesi vakit alabilmektedir.

Hata yönetiminde hataların izlenmesi ve loglanması kritik bir öneme sahiptir. Loglamanın temeli dosya tabanlı loglama olarak bilinmektedir. Bu loglama çeşidinde yaşanan durumlar bir dosya içerisinde tutulur ve büyük ölçekli sistemler için çok tercih edilmemektedir. Kullanıcıların verileri üzerinde yaşamış oldukları hataların tutulması için veritabanı loglaması karmaşık sistemler için önemlidir. Logların incelenmesiyle sorguların daha hızlı ve verimli olması sağlanabilmektedir. Sistem üzerinde yaşanan sorunların sistem üzerindeki yükü azaltması için hataların uzak bir sunucu üzerinde loglanması yapılabilmektedir. Ayrıca sunucu üzerinde olmazsa olmaz olarak nitelendirilen fonksiyonların hata vermesi durumunda loglanması sağlanır ve yaşanan hatanın anlamlandırılmasına yardımcı olur. Sistem üzerinde alınan hataların loglarının toplandığı ve analizinin sağlandığı Elasticsearch, Logstash, Kibana, Splunk ve Fluentd gibi araçlar bulunmaktadır. Bu araçlar, verilerin görselleştirilmesi, depolanması ve işlenmesini sağlamaktadır.

Monolitik mimariden mikroservis mimariye geçiş sırasında, altyapıların kontrol edilmesi için yeni yaklaşımlara ihtiyaç duyulmuştur. Servislerin ayrışmasıyla birlikte karmaşıklık artmış ve hataların izlenmesinin esnek çözümleri henüz mevcut değildir. Bu kapsamda, dağıtık bileşenlerde yaşanan problemleri çözümleyebilmek için Prometheus, Elastics APM, Grafana gibi sistemler kullanılabilmektedir.

Sistem üzerinde yaşanan problemlerde bir servisin başka bir servisi etkilememesi gerekmektedir. Örneğin, e-ticaret uygulamalarında kullanıcılar ürünlerin listesini görüntüleyip, sepete eklerken bakılan ürünlere ait veriler saklanması ya da sepete ürünün eklenemediği durumlarda hata mesajının döndürülmesiyle her servis kendisine ait durumu inceler, izolasyon sağlanır. Bu örnekte olduğu gibi hata izolasyonunun sağlanması kullanıcıların deneyimlerini de olumlu yönde etkilemektedir.

Sistem kesintilerinde yaşanacak olan hataları en az kayıp ile sağlamak gerekmektedir. Servislerde hata meydana geldiğinde kendiliğinden devre dışı bırakılır ve etkilenmemiş olan servislerin çalışmasına devam etmesi beklenmektedir. Servisin yaşadığı hatanın sonlandığı durumda tekrar devreye alınır. Hata meydana geldiği sırada çalışmasını durduran servisin yerine alternatif çözümler sunulması gerekmektedir.



Şekil 22. Mikroservis Mimari'de Devre Kesici Şeması

Kaynak: Yazar tarafından oluşturulmuştur.

Sistem üzerinde geri alma durumları belirli stratejiler ile sağlanmaktadır. Örneğin A/B testi ve Gradual Rollout (Aşamalı Dağıtım) gibi stratejiler uygulanmaktadır. Bu stratejiler ile güncellenmiş olan içerik ve güncellenmemiş olan içerik farklı kullanıcı gruplarına sunulur. Bu grupların kullanımı sırasında sistemin performansı test edilmektedir. Hata durumunda geliştirmeler geri alınmaktadır. Risklerin azaltılması için bu senaryolar sıklıkla uygulanmaktadır. Blue/Green Deployment (Mavi-Yeşil Versiyon) olarak adlandırılan strateji ise eski ve yeni versiyon olarak yayınlanır. Sistem izlenmeye başlanır ve hatasız çalışması durumunda yeni versiyon kullanılması yaygınlaştırılmaya başlanmaktadır. Maliyeti yüksek bir çalışma olduğundan dolayı çok tercih edilmemektedir.

Mikroservis mimarinin kullanımının günden güne artmasıyla birlikte otomatik hata yönetimi üzerinde çalışmalar gitgide artış göstermektedir. Örneğin, e-ticaret uygulamalarının sipariş modülünü ele aldığımızda otomatik hata tespit araçlarıyla logların izlenmesi sağlanır. Siparişlerin işlenmediği an işlemin başarısız olduğu sonucunu ilgili ekiplere uyarı olarak gönderilebilmektedir. Hızlı müdahaleyi arttıran bu araçlarla işlemlerin kesintiye uğraması minimize edilmiş olur. Bu tarz sistemlerde sıklıkla alınan hataların analizinin sağlanması ile sürekli olarak iyileştirme sağlanmaktadır. Hata analizi ile birlikte performans sorunu, yedek servise geçişin hızlandırılması gibi konular ele alınmaktadır.

2.2.5.1. Hata Yönetimi Kütüphanesi

Mikroservis mimarinin günden güne yaygınlaşması ile birlikte uygulamalarda anlık olarak yaşanan hatalara müdahale edilmesi gerekliliği kendini göstermektedir. Mikroservis mimari'de hata yönetimi kütüphanesi olarak .NET platformunda yazılım geliştiricilerin geri bildirimlerinin iyileştirilmesini sağlamak üzere oluşturulmuş bir kütüphanedir. Hata yönetiminin ve hataya karşı gösterilecek toleransın otomatikleştirilmesini sağlamaktadır. Kütüphanenin kullanılması hata yönetimi için önemlidir.

Kütüphanenin kullanılabileceği birçok sektöre uygun uygulamalar bulunmaktadır. Örneğin bir e-ticaret firmasını ele alalım; müşterilerin geri bildirimleri doğrultusunda, ürün yada hizmetlerin geri dönüşlerinin yapılması ve analizlerinin yapılmasını kolaylaştırmaktadır. Platformun kullanımı sırasında satın alma işleminde bir sorun yaşandığında geri dönüşün kütüphanenin sağlamış olduğu imkanlar doğrultusunda kontrol edilmesi ve sorunun giderilmesi sağlanabilmektedir. Kullanıcıların işlemler sırasında hata alması durumunda karşılaştıkları senaryo kapsamında güvensiz hissedip tekrar o platform üzerinden alışveriş yapma eylemini gerçekleştirmekten vazgeçebilirler. Envanter yönetiminde oluştur, oku, güncelle, sil (CRUD) operasyonlarının kullanılması sırasında yaşanan bir hata sonrası müşterilere gösterilen ürünlerin stok vb. servisler ile haberleşememesi esnasında yaşanan hataya dair müşterilere bildirim gönderilmesi gerekmektedir. Ayrıca alternatif ürünlere yönlendirilmesi veya indirim sağlanılması gibi telafi edici önlemlere başvurulması müşteri memnuniyetinin korunmasına katkı sağlamaktadır. E-ticaret firmaları, kullanıcı deneyimini iyileştirmek ve ürün performansını karşılaştırmak için geri bildirimleri etkin bir şekilde yönetmesi gerekmektedir. Potansiyel iyileştirme alanlarının belirlenebilmesi için ürün üzerinden yapılan analizlerde sistemin ürün gösteriminden, kargonun iletilmesine kadar uçtan uca bütün servislerin doğru veriler ile sürekli aynı şekilde çalışıyor olması gerekmektedir. Piyasa şartlarının gereği olarak rekabet avantajı ve pazar payının artışının sağlanması için hata yönetimine ihtiyaç duymaktadır. Sistemin işleyişi sırasında alınan hataları arayüz üzerinde oluşturulabilecek olan anket gibi, değerlendirme butonları gibi içerikler ile destekleyerek müşterinin platforma olan bağlılığını arttırmak ve markanın itibarının iyileştirilmesi sağlanılması gerekmektedir.

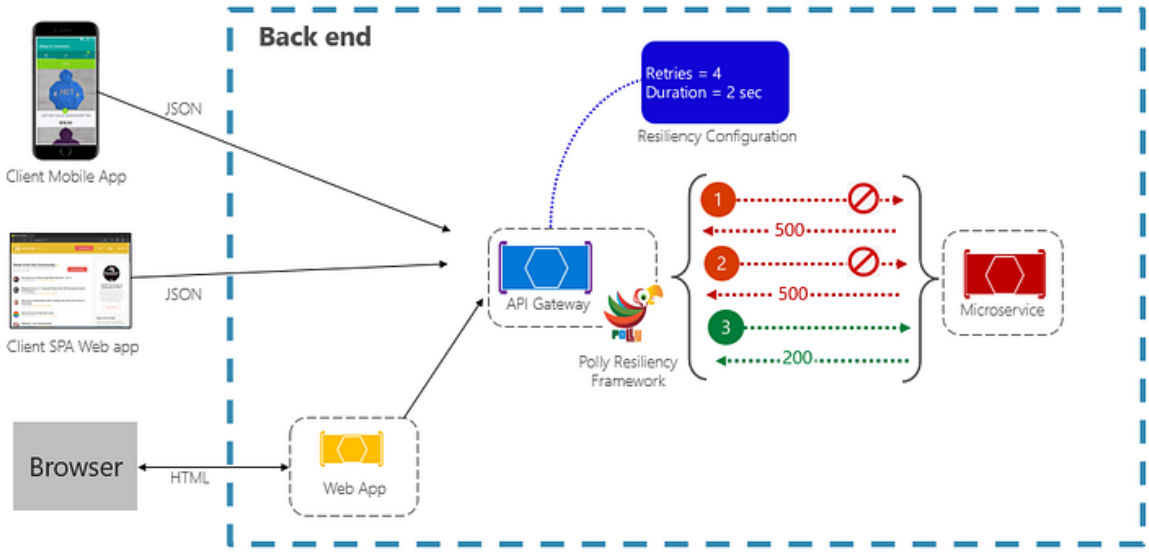
Polly kütüphanesinin kullanımı ile birlikte yazılımların güvenlik problemlerine de katkı sağlanmaktadır. Güvenlik açıkları için tek başına kullanılması yeterli olmamakta fakat kullanıldığı kısımlara uygun olarak dayanıklılığı arttırmaktadır. Başarılı şekilde uygulanmış olan kütüphane, ağ üzerinde yaşanan bağlantı sorunları kapsamında hizmet süresindeki artışın engellenmesini sağlayarak saldırıların oluşmasını zorlaştırmaktadır. Sistemin kullanımı sırasında servislere erişilmediği durumda otomatik olarak tekrar tekrar erişmeyi sağlayabilecek olan politikası ile kesintiyi engellemek sistemin istismar

edilmesine izin vermemektedir. Hataların izlenebilmesi ve kaydedilmesi ile güvenlik olaylarını kontrol etmeyi kolay hale getirmektedir. Yaşanan hata durumlarında uyarı göndermesi ile müdahale etmeyi daha mümkün kılmaktadır. Ayrıca, aşırı yükleme saldırıları yaşanması ihtimaline karşın sistem üzerindeki baskıyı azaltarak isteklerde sınırlama getirebilmektedir.

Resiliency Pattern, hataların üstesinden gelerek esneklik olarak çevrilmektedir. Bu Pattern ihtiyaç doğrultusunda istenilen uygulama içerisinde kullanılıp hatalar üzerinde daha efektif olarak çözüm sağlanmasına katkıda bulunmaktadır. Sistemin çalışabilirliğinin sağlanması, sistemde oluşabilecek olan hataların önlenmesi, hizmet kesintilerin yaşanmaması, uygulamaların sürekli çalışır vaziyette istekleri bekliyor olması gibi dayanıklılığın artışını hedeflemektedir. Polly kütüphanesinin içerisinde bulunan bu pattern hata yönetiminde kullanılabilecek önemli politikalara sahiptir.

2.2.5.1.1. Retry Pattern

Retry yönteminin kullanımı ile yapılan bütün isteklerin sonucunun hatalı dönmesi durumunda tekrar deneyebilmemize imkan sağlamaktadır. İstek gönderdiğimiz ve sonucunda dönmesini beklediğimiz servisin küçük kesintiler yaşanması durumunda otomatik olarak düzeltilebileceği bir sistem olarak kullanılır ve belirli aralıklar belirterek ardı ardına işlem yapılabilmektedir.



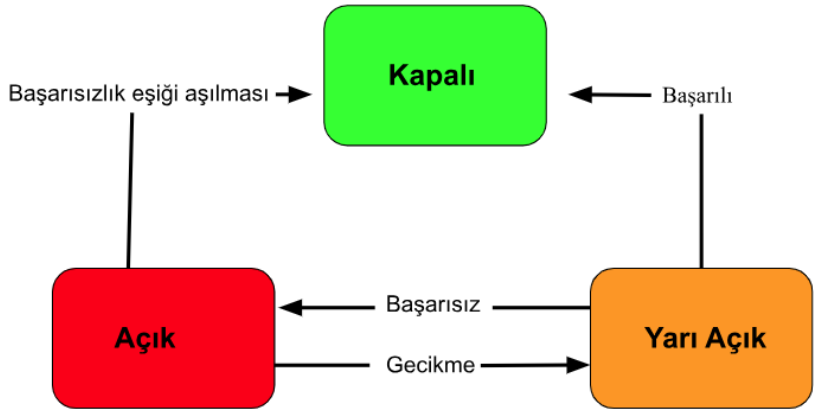
Şekil 23. Retry Pattern Şeması

Kaynak: Microsoft. (2024). *Application resiliency patterns*. 10 Şubat 2024 tarihinde https://learn.microsoft.com/en-us/dotnet/architecture/cloud-native/application-resiliency-patterns adresinden edinilmiştir.

Retry pattern kullanımında, tek bir hata, koşullu hata, birden fazla hata tipine göre uygulamanın çalışması sağlanabilmektedir. Başarılı bir sonuç alıncaya kadar tekrarlayabilir ya da bekleme süresi konularak işlemi başlattıktan sonra tekrar denemesi için bir aralık bırakılabilmektedir.

2.2.5.1.2. Circuit Breaker Pattern

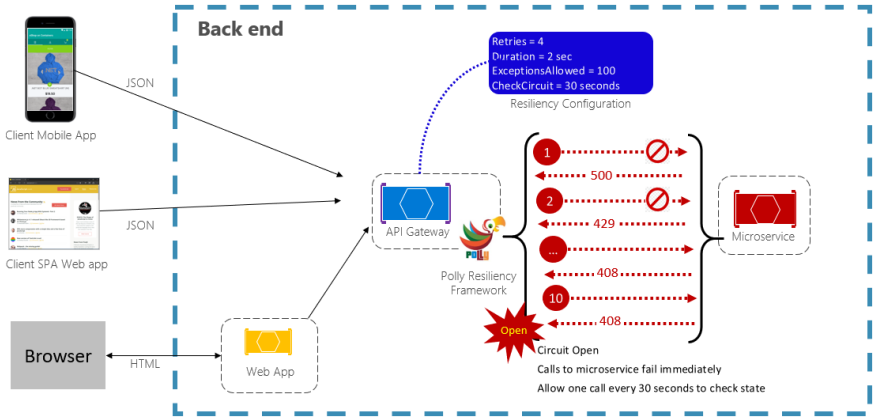
Circuit Breaker kullanımında, hatalı olan işlemlerin tekrar denenmesi durumunda bazen bir döngü içerisinde ne zaman sonu olacağını bilinmez şekilde hareket edebilmektedir. Bu tarz durumlarda bu patternin kullanılmasıyla belirli bir süre beklemesinin ardından sorguyu durdurma işlemini sağlamaktadır. Belirli olan hata tiplerinde beklenmesinin ardından hatanın durumunda değişiklik meydana gelmesi ihtimaline karşın alınabilecek aksiyonlarda belirlenebilmektedir.



Şekil 24. Circuit Breaker Durum Şeması

Kaynak: Yazar tarafından oluşturulmuştur.

Circuit Breaker kapalı olduğu durum ile yapılan bütün isteklerin geçmesine imkan sağlamaktadır. Yapılan isteklerin başarısız olarak gerçekleşmesinin ardından kendini açık konuma getirerek belirlenen süre boyunca istekleri iletmemektedir. Ardından tekrar deneme yaparak başarılı olunması durumunda tekrar kendini kapatır. Yarı açık durumunda isteklerin dinlenmesi ve oluşan senaryoya göre durumunun belirlenmesi sağlanmaktadır. Zaman aşımlarının önlenmesi ve sunucuya aşırı yüklenilmesi durumlarından kaçındırmaktadır.



Şekil 25. Circuit Breaker Şeması

Kaynak: Microsoft. (2024). *Application resiliency patterns.* 10 Şubat 2024 tarihinde https://learn.microsoft.com/en-us/dotnet/architecture/cloud-native/application-resiliency-patterns adresinden edinilmiştir.

2.2.5.1.3. Fallback Pattern

Fallback pattern kullanımında hata alınması durumunda tekrar deneme politikalarından farklı olarak başka bir servise yönlendirilmesini sağlamaktadır. Farklı bir servisin tetiklenmesi ile karşılaşılan hataya karşı çözüm sunulmuştur. Bir servisten gelen hata durumunun başka bir serviste hata durumunun kontrolünün sağlanması ile başarısız durumunun önüne geçilmiş olur. Bu şekilde kullanıcı deneyiminin iyileştirilmesine ve uygulama kullanımının artmasına katkı sağlar.

Örneğin, e-ticaret uygulamasında araçların satışının yapıldığı bir platformda, kullanıcıların ürünleri tanıtmak için fotoğraf yüklenmesi zorunlu tutulmuştur. Sistem tarafından fotoğraf yükleme servisinin çalışmaması durumunda, boş bir resmin otomatik olarak yerleştirilmesini sağlayabilmektedir.

2.2.5.1.4. Timeout Pattern

Timeout pattern, uygulama üzerinde istek gönderilen servisin zaman aşımına uğraması durumunda uygulanabilecek stratejileri içeren patterndir. Bu stratejiler ikiye optimistik ve pesimistik olarak ayrılmaktadır. Eşzamanlı yapılan işlemleri iptal etmek veya sonlandırmak için anlamına gelen 'CancellationToken' optimistik durumda asenkron işlemlerde kullanılan timeout durumudur. Pesimistik durum ise CancellationToken ihtiyacı olmadan senkron işlemlerde timeout durumunu yönetmemizi sağlamaktadır. Yapılan isteklerin süresi dolduğunu hata göstererek servis sonucunu beklemeden sistemin devam etmesine imkan sağlamaktadır.

Polly kütüphanesi, kullandığı patternler ile uygulamaların güvenliğinin, sağlamlığının ve hata toleranslarının artırılmasına katkı sağlamaktadır. Beklenmedik hataların yaşandığı durumları uygun şekillerde yanıtlayarak çözüme kavuşturabilmektedir. Kod içerisindeki tekrarın azalması ile birlikte karışıklığı önler. Sağlamış olduğu hızlı çözümleri ile birlikte gerekli olan uygulamalarda hata yönetiminde kullanılması için tercih edilmesi gereklidir. Yazılım geliştiricilerin kod tarafındaki kullanışlılığı ile uygulamanın bakımının kolaylaştırmasına ve güncelleştirilmesine de katkı sağlamaktadır.

3.UYGULAMA

Bu tez kapsamında, mikroservis mimarisinde hata yönetiminde e-ticaret uygulamalarında dikkat edilmesi gerekenler ile müşterilerin deneyiminin önemli ölçüde iyileştirilmesi hedeflenmektedir. Hata yönetim kütüphanelerinin geniş çaplı kitlelere hitap eden e-ticaret uygulamaları üzerinde kullanılması ile birlikte otomatik iyileştirmeler, hataları tekrar denenebilmesi, servislerden hızlı geri dönüşler sağlanması ile müşteri memnuniyetini arttırarak kesintisiz şekilde hizmet sunabilecek e-ticaret uygulamalarının müşterilerine sağladığı fayda kritik olarak öne çıkmaktadır.

3.1. Uygulamada Kullanılan Teknoloji ve Kütüphaneler

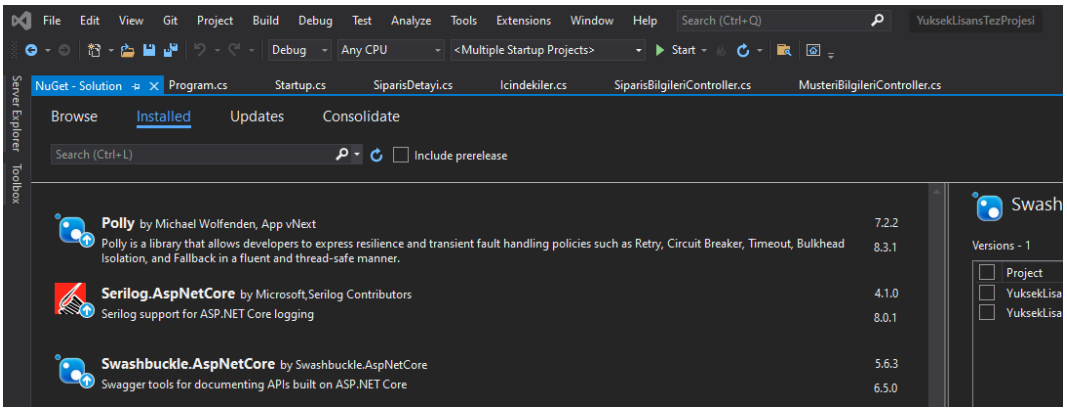
- Visual Studio 2019: Yazılım geliştirme süreçlerini kolaylaştırmak adına Microsoft tarafından geliştirmiştir. Kod düzenleme ve derleme gibi işlemlerin yapılmasını desteklemektedir.

- ASP.NET Core Web API: HTTP protokolü üzerinde WEB API'lerin geliştirmesi için ASP.NET Core tabanlı platform bağımsız olarak çalışmaktadır.

- Serilog.AspNetCore: .NET Core uygulamalarında yapılan işlemleri kaydetmek, oluşan uyarı ve hatalar hakkında sorunların izlenmesini sağlamak için kullanılan bir kütüphanedir.

- Polly: Hataların işlenmesi, yönetilmesi, yeniden denenebilmesi ve geri alınabilmesi gibi konularda uygulamaların daha hata yönetimi etkili şekilde sağlaması için kullanılan kütüphanedir.

- Swashbuckle.AspNetCore: ASP.NET Core projelerinde Swagger ile API'lerin belgelendirilmesi ve anlaşılmasını kolaylaştırmaktadır. API'lerin içerisinde bulunan parametreler ve verilerin test edilmesi için bir arayüz sunmaktadır.

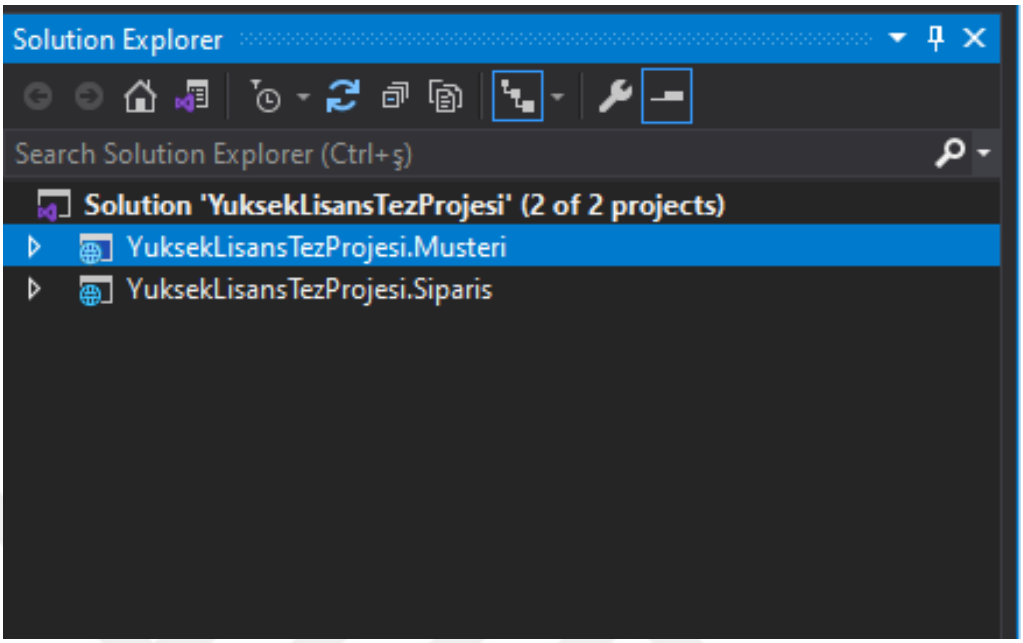


Şekil 26. Uygulamada Kullanılan Kütüphaneler

Kaynak: Yazar tarafından oluşturulmuştur.

3.2. Uygulama Mimarisi ve Modüllerin Tanımı

Uygulama mikroservis mimarisi üzerine kurulmuştur. Uygulama 2 adet API'den oluşmaktadır. Bu API'ler Sipariş ve Müşteri olarak iki farklı uygulama şeklinde ele alınmaktadır. Kendileri bağımsız olarak olarak görevleri yerine getirirler. Sipariş API'si, mevcut müşterilerin kontrolünü sağlayarak sipariş oluşturulmasını sağlar. Siparişlerin oluşturulması esnasında yaşanan problemlere dair aldığı önlemleri metodlar halinde içerisinde barındırır. Müşteri API'si müşteri bilgilerinin yönetilmesi için oluşturulmuştur. Müşteriye ait olan veriler ile siparişin oluşturulması sağlanmaktadır. Bağımsız olarak çalışan bu API'lerde Polly kütüphanesi ile oluşan hatalar ele alınır ve bu hataların yönetilmesine dair uygulama üzerinde dayanıklılığın artırılmasına destek sağlanır.

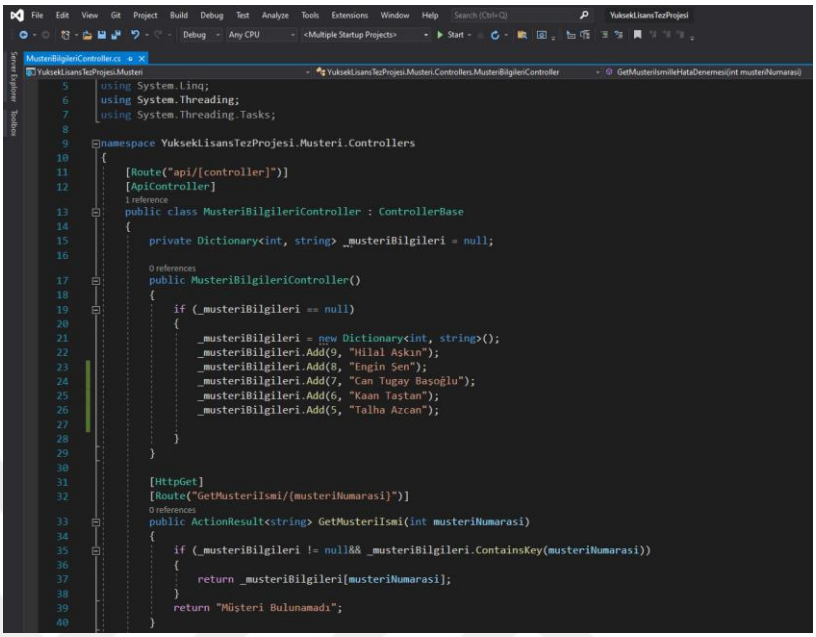


Şekil 27. Mikroservis Projesi Mimarisi

Kaynak: Yazar tarafından oluşturulmuştur.

3.2.1. Müşteri API

Uygulama üzerinde müşteri bilgilerini içeren bir sınıf oluşturulmuştur. Bu sınıf içerisinde müşteriye ait müşteri numarası alanı ve müşterinin ad soyad alanı bulunmaktadır. Örnek isimler üzerinden 6 farklı kullanıcının sisteme girişi sağlanmaktadır. Müşterilerin bilgilerinin sistem tarafından gösterilmesi için HTTP GET isteği ile müşterinin id numarasını göndererek, bu id numarasına karşılık gelen müşteri ismini Swagger arayüzünde göstermektedir. Swagger arayüzü üzerinden API'lere HTTP isteği yapılabilmektedir. Mevcut API üzerinde yapılan istekler ile sağlamış olduğu özellikler görüntülenebilmektedir.

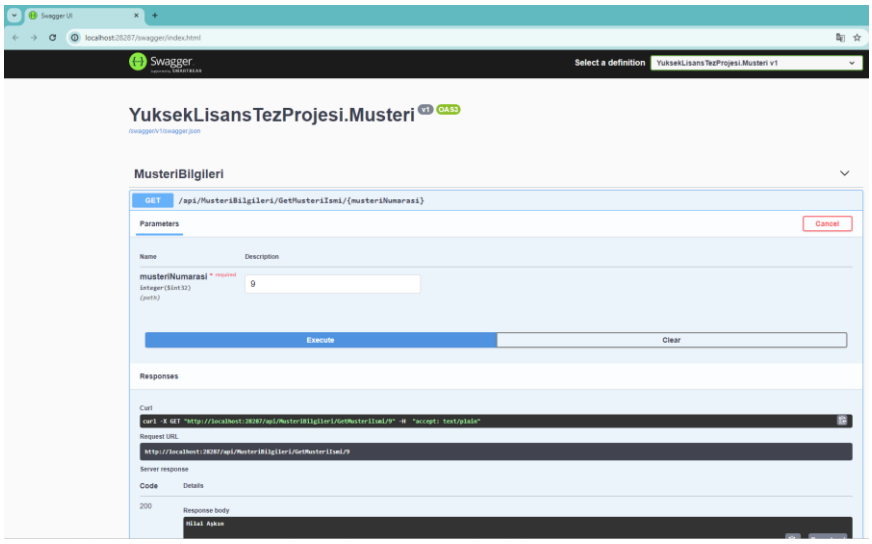


```
using System.Linq;
using System.Threading;
using System.Threading.Tasks;

namespace YuksekLisansTezProjesi.Musteri.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    1 reference
    public class MusteriBilgileriController : ControllerBase
    {
        private Dictionary<int, string> _musteriBilgileri = null;

        0 references
        public MusteriBilgileriController()
        {
            if (_musteriBilgileri == null)
            {
                _musteriBilgileri = new Dictionary<int, string>();
                _musteriBilgileri.Add(9, "Hilal Aşkın");
                _musteriBilgileri.Add(8, "Engin Şen");
                _musteriBilgileri.Add(7, "Can Tugay Basoğlu");
                _musteriBilgileri.Add(6, "Kaan Taştan");
                _musteriBilgileri.Add(5, "Talha Azcan");

            }
        }

        [HttpGet]
        [Route("GetMusteriIsmi/{musteriNumarasi}")]
        0 references
        public ActionResult<string> GetMusteriIsmi(int musteriNumarasi)
        {
            if (_musteriBilgileri != null&& _musteriBilgileri.ContainsKey(musteriNumarasi))
            {
                return _musteriBilgileri[musteriNumarasi];
            }
            return "Müşteri Bulunamadı";
        }
```

Şekil 28. Müşteri Bilgileri Metodu

Kaynak: Yazar tarafından oluşturulmuştur.

Uygulama içerisinde bulunan metodlar ile Swagger aracılığıyla API'leri daha iyi anlamamıza katkı sağlamaktadır. API'leri bu kapsamda test edebilir ve geri dönüş cevaplarının alınmasını daha kolay hale getirmektedir.



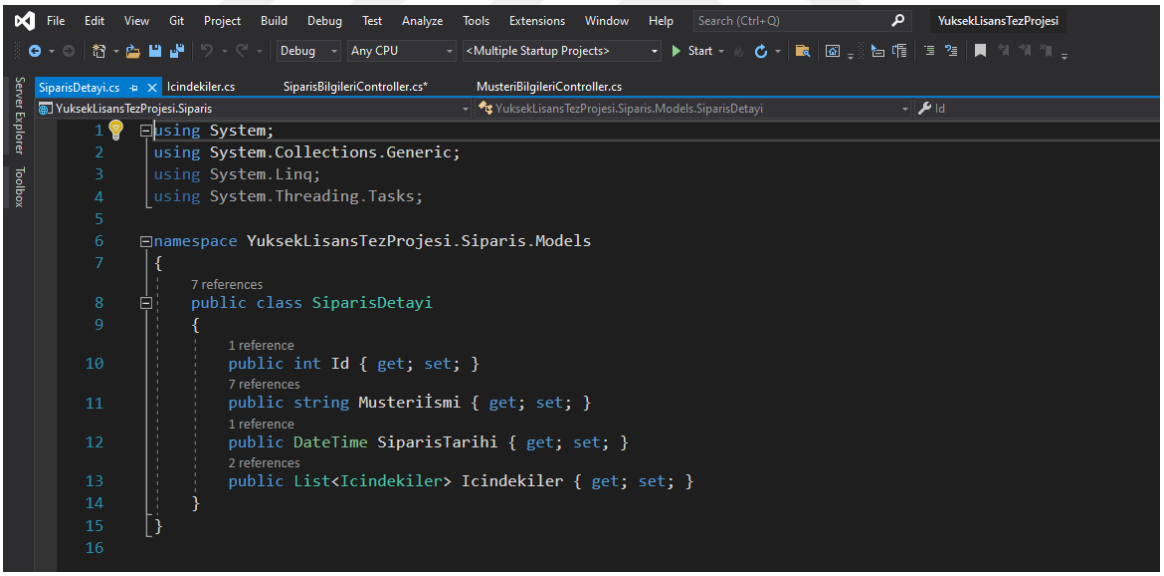
Şekil 29. GetMusteriIsmi Servisinin Çalıştırılması

Kaynak: Yazar tarafından oluşturulmuştur.

Uygulamanın hata ile karşılaşmadan çalışmış olduğu durumda Swagger arayüzünde "**/api/MusteriBilgileri/GetMusteriIsmi/{musteriNumarasi}**" endpointine kodlama tarafında varsayılan olarak belirtilmiş olan müşteri numarası gönderilerek kişinin bilgilerinin alınması sağlanmaktadır. Hata yönetimin buradaki önemi HTTP durum kodlarından alınan geri dönüş değerine göre uygulama yaşanan problemi ifade etmektedir.

3.2.2. Sipariş API

Uygulama geliştirme sürecinde müşteri bilgilerinin ve siparişe ait olan bilgilerin alınması için API oluşturulmaktadır. Bu API içerisinde barındırdığı verileri bir model içerisinde bulundurmaktadır. Siparişin ayrıntılarını içeren bu sınıf, siparişi veren müşterinin bilgilerini ve sipariş tarihi gibi içerikler servisin içerisinde bulunan verileri barındırmaktadır.

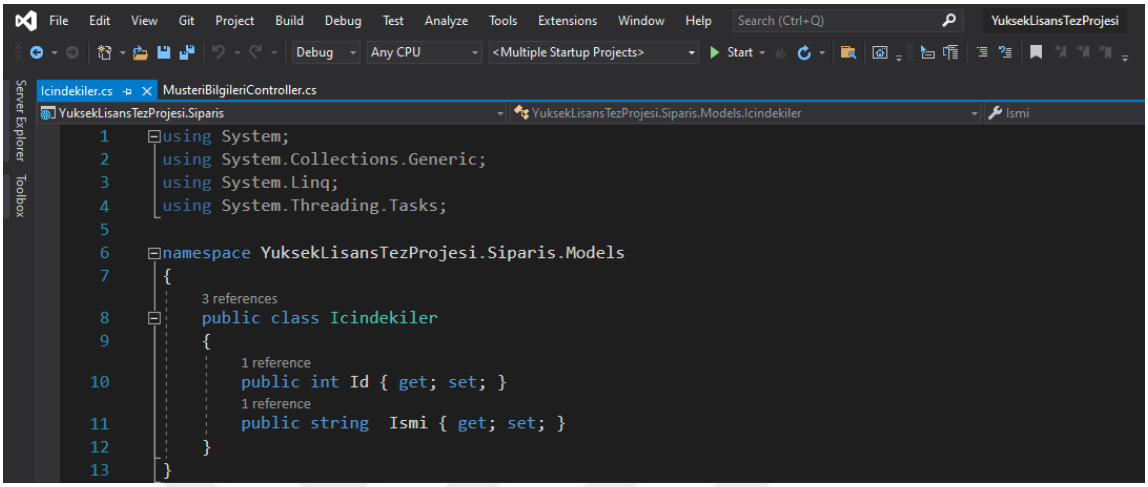


Şekil 30. Sipariş Detayi Sınıfının Eklenmesi

Kaynak: Yazar tarafından oluşturulmuştur.

Sipariş detayı içerisinde bir müşterinin birden fazla siparişi olabilmektedir. Bu sebeple siparişleri sepet olarak ekleyebilmektedir. Bir sipariş kendine ait eşsiz bir değer ile tutulmaktadır. Bu eşsiz değer sipariş numarası olarak adlandırılırken siparişin içerisine

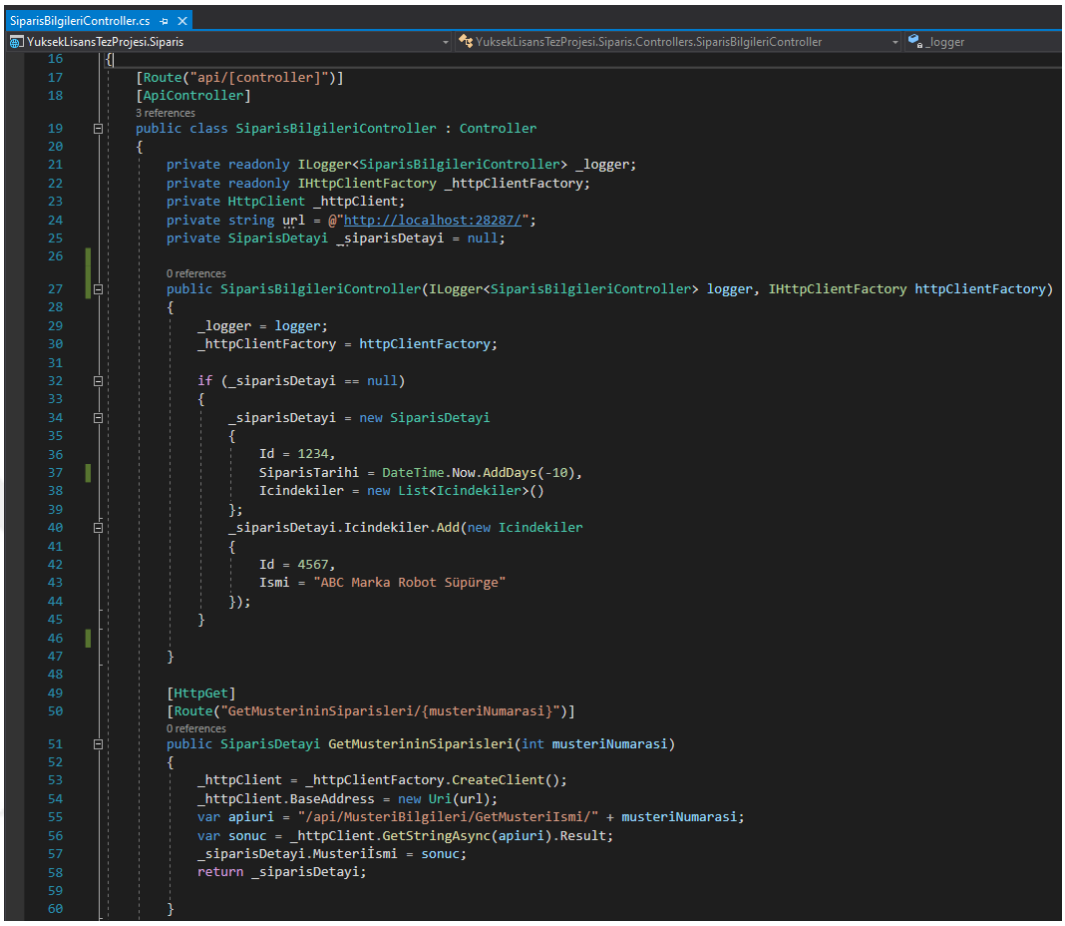
eklenen her ürün içindekiler class'ı içerisinde id numarası ve ürünün ismi ile yer almaktadır.



Şekil 31. İçindekiler Sınıfının Eklenmesi

Kaynak: Yazar tarafından oluşturulmuştur.

E-ticaret uygulamalarında yaşanabilecek birden çok tipte senaryo mevcuttur. Sipariş verilmesi sırasında stokta olmayan bir ürün sipariş edilebilir. Yapılan güncellemeler sırasında sipariş işlemi tamamlanmadan önce fiyatta değişiklik yaşanabilir. Ödeme işlemi yapılırken yetersiz bakiye ile karşılaşılabilir. Ödeme alındıktan sonra kargoya iletilmesi ve teslimatının sağlanması konusunda hatalar yaşanabilir. Müşteriler teslimat adreslerini hatalı girebilirler ve iptal etmeye ihtiyaç duyabilirler. Bu sebeple, ürünlerin kendine ait id'si işlem yapılmaktadır. Her bir ürün kendi sipariş id'si ile birlikte değerlendirilmektedir. Sipariş işlemi müşteri memnuniyeti açısından süreci doğrudan etkilemektedir. Ürün açıklamaları, resimleri, müşterilerin yorumları ile geri dönüşler alınması müşterilerin uygulamaya olan güvenini de arttırmaktadır.



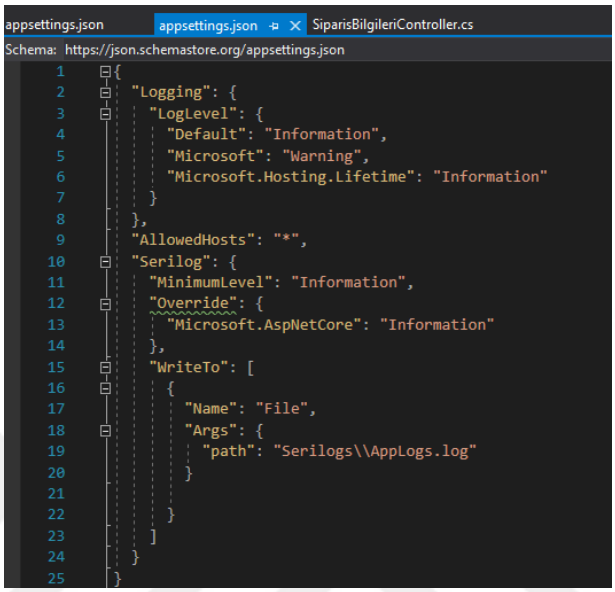
Şekil 32. Sipariş Bilgileri Metodu

Kaynak: Yazar tarafından oluşturulmuştur.

3.3. Uygulama İçerisinde Hata İzleme Kütüphanesi Entegrasyonu

E-ticaret uygulamalarında yaşanalabilecek sorunlara en hızlı şekilde müdahale ederek çözümlenmesi için hata izleme kütüphanelerinden faydalanılması gerekmektedir. Bu tez içerisinde Serilog.AspNetCore kütüphanesi ile hataların izlenilmesi, güvenlik açıklarının tespitinin sağlanması, hataların takibinin yapılabilmesi ve uygulamanın performansının iyileştirilmesi sağlanmaktadır.

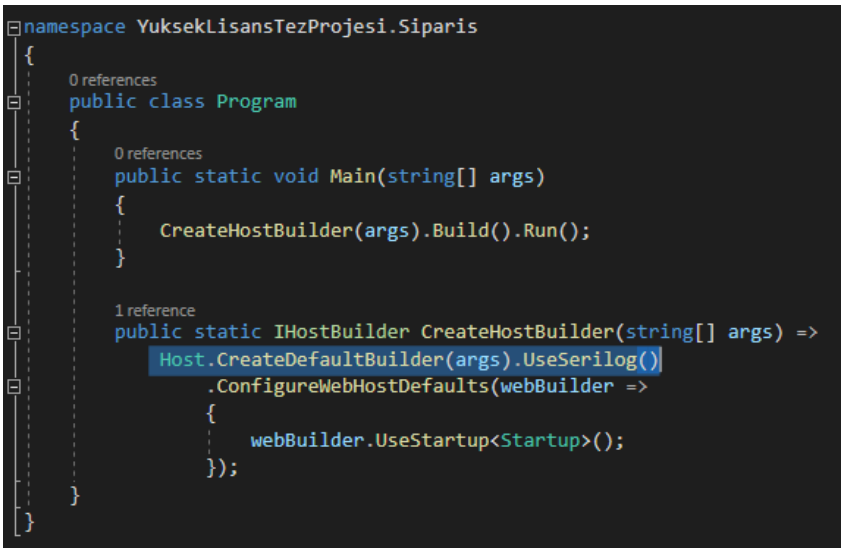
Uygulama içerisinde ilgili kütüphanenin eklenilmesi ile birlikte appsetting. json dosyası içerisinde yapılandırılmasının sağlanması gerekmektedir.



Şekil 33. Serilog Kütüphanesinin Yapılandırılması -1

Kaynak: Yazar tarafından oluşturulmuştur.

Uygulama içerisinde ilgili kütüphane program.cs dosyasına eklenmiştir.



Şekil 34. Serilog Kütüphanesinin Yapılandırılması -2

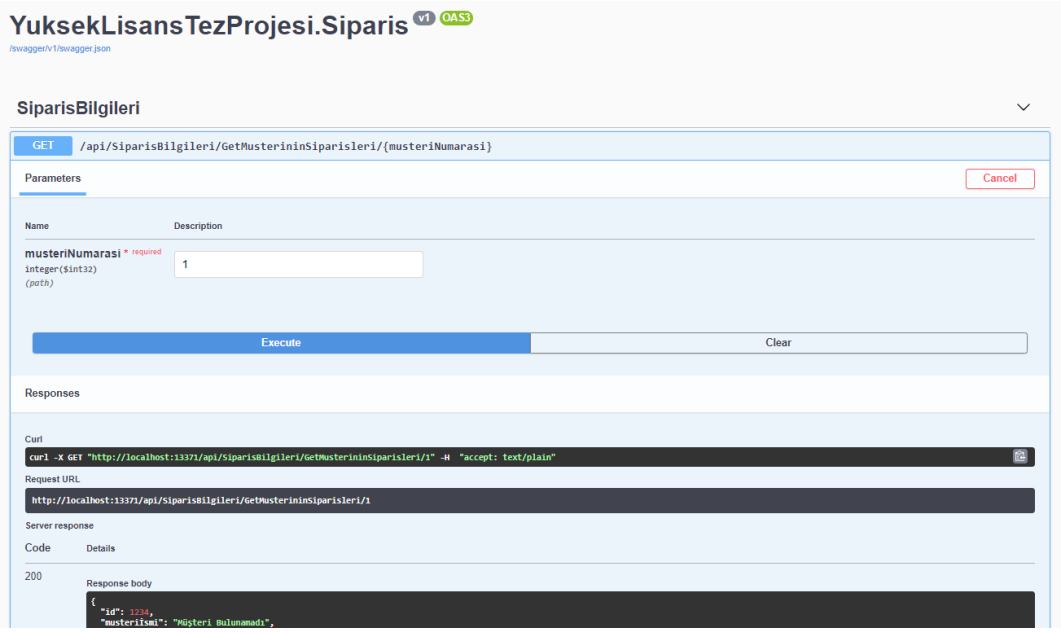
Kaynak: Yazar tarafından oluşturulmuştur.

```
namespace YuksekLisansTezProjesi.Siparis
{
    2 references
    public class Startup
    {
        0 references
        public Startup(IConfiguration configuration)
        {
            Configuration = configuration;
            Log.Logger = new LoggerConfiguration()
                .ReadFrom.Configuration(configuration)
                .CreateLogger();
        }
```

Şekil 35. Serilog Kütüphanesinin Yapılandırılması -3

Kaynak: Yazar tarafından oluşturulmuştur.

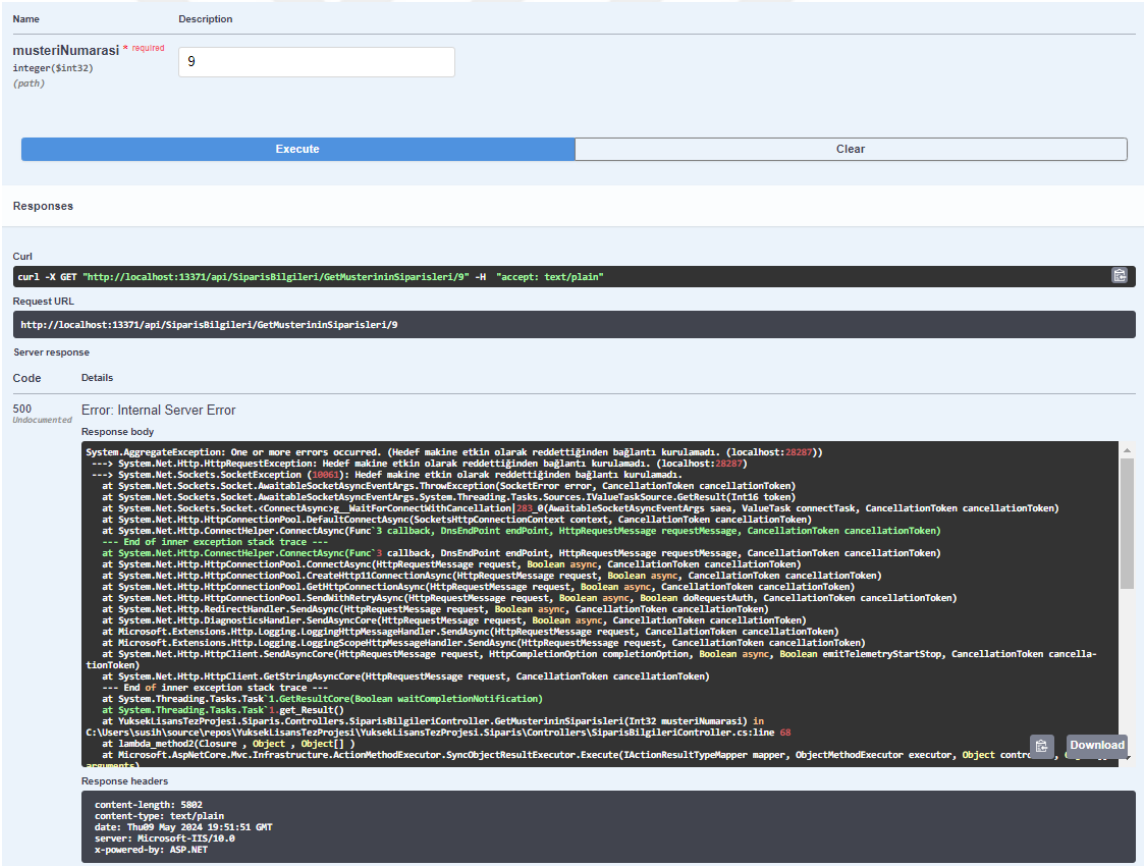
Serilog.AspNetCore kütüphanesi eklenilmesinin ardından test edilmesi sırasında **"/api/SiparisBilgileri/GetMusterininSiparisleri/{musteriNumarasi}"** müşteriye ait olan siparişlerin listelendiği endpoint çalıştırıldığında müşteri numarası parametresi müşteri API'sine gönderilir ve geri dönüş alındığı durumda müşteriye ait olan siparişler listelenmektedir.



Şekil 36. Müşteriye Ait Siparişlerin Listelenmesi

Kaynak: Yazar tarafından oluşturulmuştur.

Serilog.AspNetCore kütüphanesi sipariş API'sine bağlanmakta sorun yaşamadığı fakat müşteri API'sinden geri dönüş alamadığı durumlarda HTTP durum kodu olarak 500 geri dönüş değerine sahip olmaktadır. Başarılı bir şekilde API bağlantısı kurulamadığı durumda bağlantı hatası ya da ağ sorunları meydana gelmiş olabilir. Eğer uygulama kimlik doğrulama yöntemi ya da yetkilendirme yöntemi kullanıyor olsaydı bu tip sorunları da göz önünde bulundurmak gerekmektedir. Yaşanan bu hata kütüphane tarafından izlenilip, hataya dair uygun alternatifler stratejileri uygulaması önem arz etmektedir. Uygulama geliştiriciler yaşanan hataların izlenelebilmesi ile sipariş sürecini etkili bir şekilde yönetebilmektedir. Aşağıdaki görselde görüldüğü gibi müşteri numarası sistem içerisinden mevcut olmasına rağmen API çalışır durumda olmadığı için hata vermektedir.



Şekil 37. Müşteri API'sine Ulaşılamadığı Hata Durumu

Kaynak: Yazar tarafından oluşturulmuştur.

3.4. Uygulama İçerisinde Hata Yönetimi ve İyileştirme Stratejileri

Hata iyileştirme kütüphanesi olarak uygulama içerisinde Polly kütüphanesi tercih edilmiştir. Bu kütüphane uygulamaya paket olarak yüklenmiştir. İki API şeklinde hazırlanmış olan projeye SiparisAPI için entegre edilmiştir. Müşteri servisinden dönüş değeri alınamadığı durumlarda dahi SiparisAPI dayanıklı hale getirilmiştir.

SiparisAPI tarafından MusteriAPI'sine yapılan bir HTTP isteğinin yanıt vermemesi yani isteğin başarısız sonucu döndürmesiyle yaşanan hatanın anlık bir durum olabilmesi ihtimaline karşın tekrar deneme yapılması gerekmektedir. Bu tekrar deneme sonsuz döngü olarak tekrarlanmaması için proje içerisinde 3 kere deneme yapılmasına göre geliştirme yapılmıştır.

SiparisAPI'si, müşteriden olumlu bir yanıt alamadığı durumda deneme sayısı tamamlandığı an istek olumsuz olarak durdurulacaktır. MüsteriAPI'sine ait rastgele bir hata durumunu belirtmek için rakamların rastgele olarak üretilen sayılar içerisinden çift sayı gelirse eğer 500 hatası alınır, tek sayı olması durumunda olumlu bir şekilde sonuçlanmaktadır. Bu metoda göre rastgele oluşan sonuca bakılarak hataların deneme yapılmaktadır. Polly kütüphanesi kullanılarak yeniden deneme sağlayan RetryPolicy ile kodlanması sağlanmıştır.

```
[HttpGet]
[Route("GetMusteriIsmiIleHataDenemesi/{musteriNumarasi}")]
0 references
public ActionResult<string> GetMusteriIsmiIleHataDenemesi(int musteriNumarasi)
{
    try
    {
        Random rnd = new Random();
        int rndError = rnd.Next(1, 11);
        if (rndError % 2 == 0)
        {
            throw new Exception();
        }
        if (_musteriBilgileri != null && _musteriBilgileri.ContainsKey(musteriNumarasi))
        {
            return _musteriBilgileri[musteriNumarasi];

        }
        return "Müşteri Bulunamadı";
    }
    catch
    {
        return StatusCode(StatusCodes.Status500InternalServerError);
    }
}
```

Şekil 38. RetryPolicy Uygulaması-1

Kaynak: Yazar tarafından oluşturulmuştur.

```
public SiparisBilgileriController(ILogger<SiparisBilgileriController> logger, IHttpClientFactory httpClientFactory)
{
    _logger = logger;
    _httpClientFactory = httpClientFactory;

    if (_siparisDetayi == null)
    {
        _siparisDetayi = new SiparisDetayi
        {
            Id = 1234,
            SiparisTarihi = DateTime.Now.AddDays(-10),
            Icindekiler = new List<Icindekiler>()
        };
        _siparisDetayi.Icindekiler.Add(new Icindekiler
        {
            Id = 4567,
            Ismi = "ABC Marka Robot Süpürge"
        });
    }
    _retryPolicy = Policy.Handle<Exception>().Retry(3);
}
```

Şekil 39. RetryPolicy Uygulaması-2

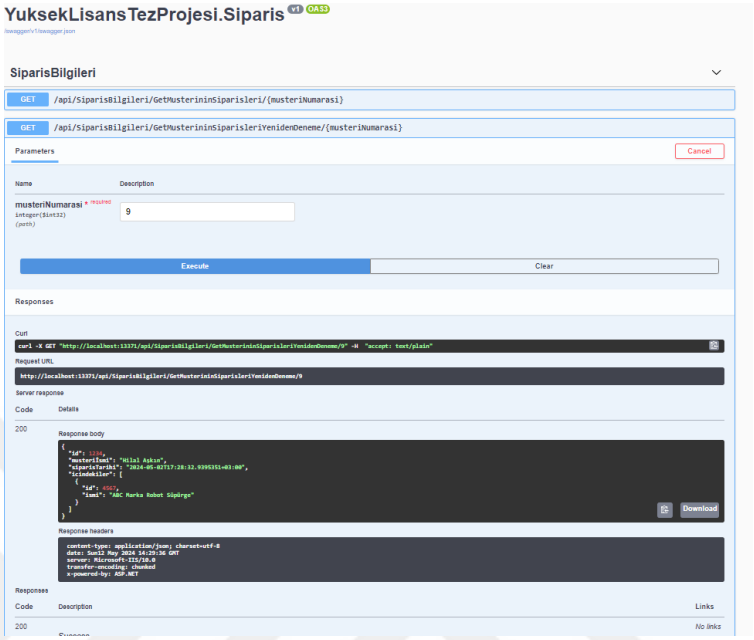
Kaynak: Yazar tarafından oluşturulmuştur.

SiparisAPI'sine içerisindeki ürün işlenmesinin ardından yeniden deneme yapacaktır. Müşterinin siparişlerinin detayını almak için metod oluşturulmuştur. Müşterinin numarasını ele alarak bu numaraya sahip olan siparişin detayını göstermektedir. İsteğin gösteriminin sağlanacağı URL belirtilmiştir. HTTP isteğinin oluşması durumunda yaşanabilecek olan hata başarılı bir sonuç dönünceye kadar 3 kez tekrarlanmaktadır. Sonucun başarılı bir şekilde alınmasının ardından siparişin detayının bulunduğu nesnenin gösterimi sağlanmaktadır.

```
[HttpGet]
[Route("GetMusterininSiparisleriYenidenDeneme/{musteriNumarasi}")]
0 references
public SiparisDetayi GetMusterininSiparisleriYenidenDeneme(int musteriNumarasi)
{
    _httpClient = _httpClientFactory.CreateClient();
    _httpClient.BaseAddress = new Uri(url);
    var apiuri = "/api/MusteriBilgileri/GetMusteriIsmiIleHataDenemesi/" + musteriNumarasi;
    var sonuc = _retryPolicy.Execute(() => _httpClient.GetStringAsync(apiuri).Result);
    _siparisDetayi.MusteriIsmi = sonuc;
    return _siparisDetayi;
}
```

Şekil 40. RetryPolicy Uygulaması-3

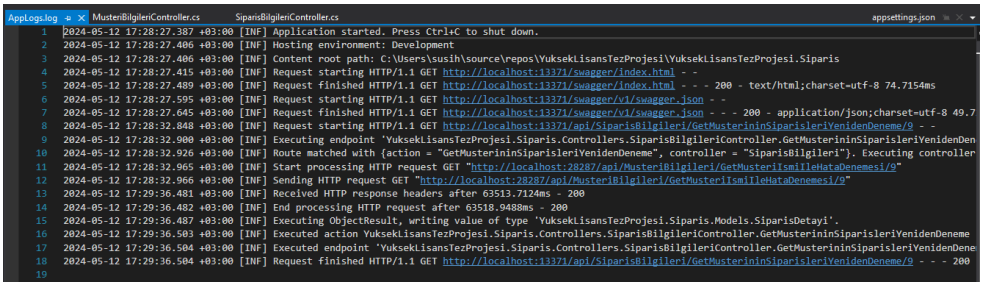
Kaynak: Yazar tarafından oluşturulmuştur.



Şekil 41. RetryPolicy Uygulaması-4

Kaynak: Yazar tarafından oluşturulmuştur.

Uygulama üzerinde iki API'ninde çalıştırmasının ardından açılan arayüz de **"/api/SiparisBilgileri/GetMusterininSiparisleriYenidenDeneme/{musteriNumarasi}"** metodunda müşterinin numarasının girilmesinin ardından logları görüntülemek için kullandığımız kütüphane ile App.log dosyasının içerisinde yapılan istekleri ve bu isteklerin kaç kere tekrar edildiği görüntülenebilmektedir.



Şekil 42. RetryPolicy Uygulaması-5

Kaynak: Yazar tarafından oluşturulmuştur.

Uygulama üzerinde yeniden deneme politikasının uygulanması ile birlikte e-ticaret uygulamalarında müşterinin numarasını göndererek, ilk yapılan istekte hata dönmesine rağmen tekrar denendiğinde başarılı bir sonuç elde edilmiştir. Bu kapsamda, sipariş verilmesi sırasında oluşan geçici hatalarının önüne geçilmesi için kullanılması gerekmektedir.

Uygulama tarafından belirlenmiş olan süre içerisinde bir servisten yanıt alınamaması durumunda tekrar deneme işlemi yapılabilmektedir. Fakat yapılan tekrar deneme işlemi sonsuz kez tekrar edemeyeceği için zaman aşımına uğraması durumunda uygulamanın kendisini durdurması ve isteği sonlandırması gerekmektedir. Siparişin verilmesi sırasında müşterinin ismine ihtiyaç duyulmaktadır. Müşteri servisinin yanıt vermemesi durumunda örnek bir senaryo üzerinde 3 dakikalık bir gecikme yaşatılmıştır.

```
[HttpGet]
[Route("GetMusteriIsmiIleGecikme/{musteriNumarasi}")]
0 references
public ActionResult<string> GetMusteriIsmiIleGecikme(int musteriNumarasi)
{
    Thread.Sleep(new TimeSpan(0, 3, 0));
    if (_musteriBilgileri != null && _musteriBilgileri.ContainsKey(musteriNumarasi))
    {
        return _musteriBilgileri[musteriNumarasi];
    }
    return "Müşteri Bulunamadı";
}
```

Şekil 43. Timeout Policy-1

Kaynak: Yazar tarafından oluşturulmuştur.

Bu gecikmenin sonucunda eğer hala yanıt alınamaz ise 30 saniye boyunca bekletilir ve başarılı bir sonuç alınmadığı için hata döndürülmektedir. Bu şekilde, uygulama sonsuz kere deneme yapmamaktadır.

```
    });
    }
    _timeoutPolicy = Policy.Timeout(30, TimeoutStrategy.Pessimistic);
}
```

Şekil 44. Timeout Policy-2

Kaynak: Yazar tarafından oluşturulmuştur.

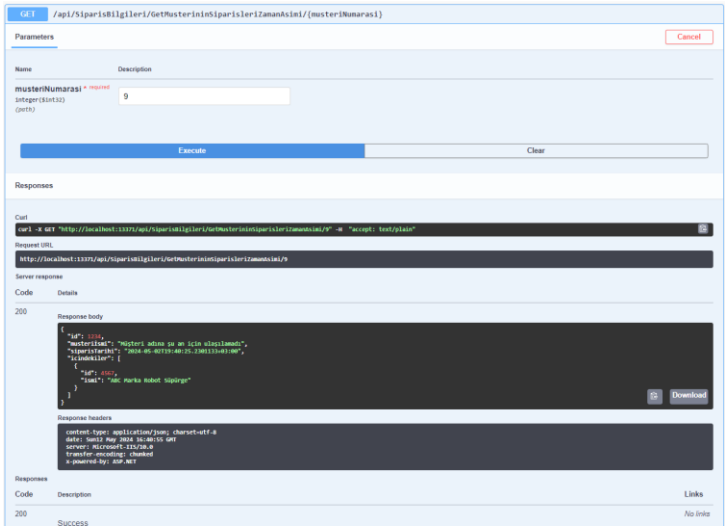
Müşteri bilgilerine ulaşılamadığı durumda kod tarafında hata durumu API üzerinde hataya dair bilgilendirme mesajı gösterilmektedir. Gösterilen hataya dair verilen kısa bilgi gerektiği durumda daha detaylı olarak analiz edilebilmektedir.

```
[HttpGet]
[Route("GetMusterininSiparisleriZamanAsimi/{musteriNumarasi}")]
0 references
public SiparisDetayi GetMusterininSiparisleriZamanAsimi(int musteriNumarasi)
{
    try
    {
        _httpClient = _httpClientFactory.CreateClient();
        _httpClient.BaseAddress = new Uri(url);
        var apiuri = "/api/MusteriBilgileri/GetMusteriIsmiIleGecikme/" + musteriNumarasi;
        var sonuc = _timeoutPolicy.Execute(() => _httpClient.GetStringAsync(apiuri).Result);
        _siparisDetayi.MusteriIsmi = sonuc;
        return _siparisDetayi;
    }
    catch (Exception ex)
    {
        _logger.LogError(ex, "Sefer Gerçekleşti");
        _siparisDetayi.MusteriIsmi = "Müşteri adına şu an için ulaşılamadı";
        return _siparisDetayi;
    }

}
```

Şekil 45. Timeout Policy-3

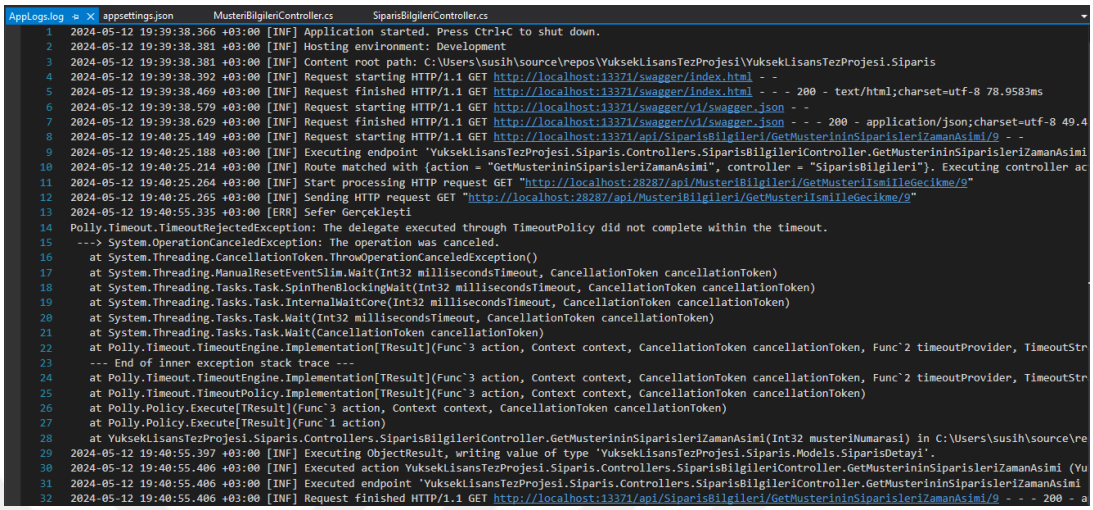
Kaynak: Yazar tarafından oluşturulmuştur.

Uygulama üzerinde iki API'ninde çalıştırmasının ardından açılan arayüz de **"/api/SiparisBilgileri/GetMusterininSiparisleriZamanAsimi/{musteriNumarasi}"** metodunda müşterinin numarasının girilmesinin ardından beklenen süreye dair zaman aşımına uğradığı yer app.log dosyası içerisinde görüntülenmektedir. Ayrıca Swagger arayüzünde müşterinin numarası girildiğinde zaman aşımı sonrası hata gösterilmiştir.



Şekil 46. Timeout Policy-4

Kaynak: Yazar tarafından oluşturulmuştur.



Şekil 47. Timeout Policy-5

Kaynak: Yazar tarafından oluşturulmuştur.

Bu kapsamda, sipariş verilmesi sırasında oluşan hataların sonsuz döngüye girmek yerine belirli bir süre denemesi yapıldıktan sonra gecikmelerin önüne geçilmesi için kullanılması gerekmektedir. Bu gecikmelerin önüne geçilmesi ile birlikte müşterilerin zaman kaybının minimum seviyeye indirilmesi hedeflenmektedir.

Sipariş servisinden müşteri servisine yapılan bir isteğin başarısız olması durumunda uygulamanın ilk aşamalarında bahsedilen yeniden deneme politikası uygulanabilir. Fakat yeniden denemeler sonucunda başarısız sonucu döndürülmesi yerine bu sonuca alternatif çözüm olarak Fallback politikası uygulanabilmektedir.

```
private readonly FallbackPolicy<string> _fallbackPolicy;
0 references
public SiparisBilgileriController(ILogger<SiparisBilgileriController> logger, IHttpClientFactory httpClientFactory)
{
    _logger = logger;
    _httpClientFactory = httpClientFactory;

    if (_siparisDetayi == null)...
     _fallbackPolicy = Policy<string>.Handle<Exception>().Fallback("Müşteri ismi bulunamadı, tekrar deneyin");
}
```

Şekil 48. Fallback Policy-1

Kaynak: Yazar tarafından oluşturulmuştur.

Yapılan isteğin sonucunda yaşanan hata veritabanı bağlantı hatası olması durumunda müşteri ismi bulunamadı şeklinde bir dönüş gösterilmektedir. Alternatif çözümler sunulması ile sipariş servisinde hataya sebep olmadan kullanıcıların beklentilerine uygun şekilde geri dönüş değeri gösterilmektedir. Kullanıcıya gösterilen dönüş mesajları ile HTTP durum kodları anlamlandırılmakta ve alternatif bir çözüm önerilmektedir.

```
[HttpGet]
[Route("GetMusterininSiparisleriniGeriDondur/{musteriNumarasi}")]
0 references
public SiparisDetayi GetMusterininSiparisleriniGeriDondur(int musteriNumarasi)
{
    _httpClient = _httpClientFactory.CreateClient();
    _httpClient.BaseAddress = new Uri(url);
    var apiuri = "/api/MusteriBilgileri/GetMusteriIsmiIleIzinHatasi/" + musteriNumarasi;
    var sonuc = _fallbackPolicy.Execute(() => _httpClient.GetStringAsync(apiuri).Result);
    _siparisDetayi.MusteriIsmi = sonuc;
    return _siparisDetayi;
}
```

Şekil 49. Fallback Policy-2

Kaynak: Yazar tarafından oluşturulmuştur.

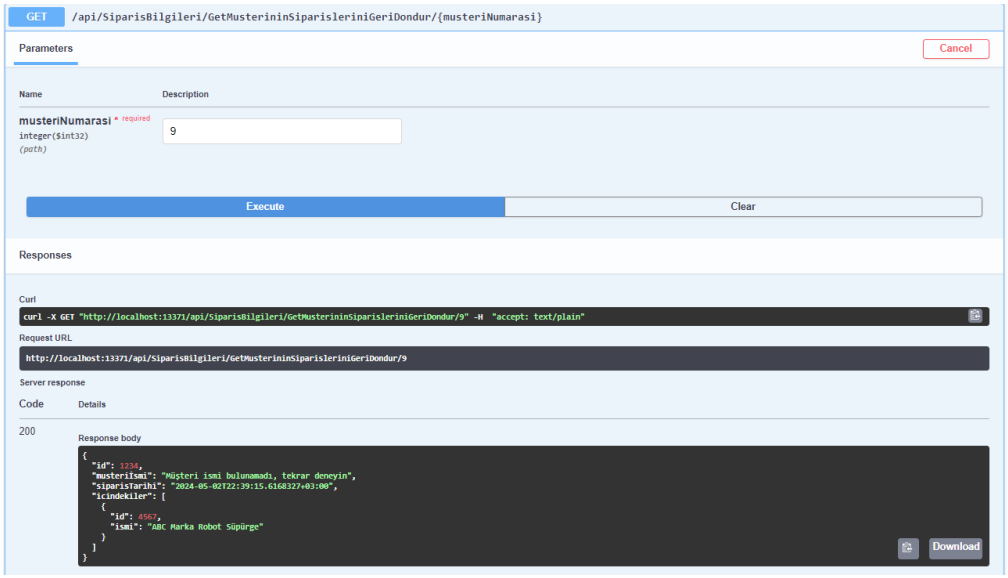
HTTP durum kodları yaşanan hata durumuna göre ayrı ayrı incelenmesi gerekmektedir. Uygulama üzerinde API'lerin çalıştırılmasının ardından açılan arayüz **"/api/SiparisBilgileri/GetMusterininSiparisleriniGeriDondur/{musteriNumarasi}"** metodunda müşterinin numarasının girilmesinin ardından hata meydana gelmiştir. Fakat hata meydana gelmesine rağmen HTTP durum kodu olarak 200 geri dönüş değeri alınmıştır. Swagger arayüzünde görüntülenen bu dönüş değeri app.log içerisinde deneme yapıldığı ve ardından geri dönüş değerinin 500 durum kodu alındıktan sonra bu politikanın uygulanması ile 200 geri dönüş değeri elde edilmiştir.

```
[HttpGet]
[Route("GetMusteriIsmiIleIzinHatasi/{musteriNumarasi}")]
0 references
public ActionResult<string> GetMusteriIsmiIleIzinHatasi(int musteriNumarasi)
{
    try
    {
        throw new Exception("Veritabanı Bulunamadı");
    }
    catch (Exception)
    {
        return StatusCode(StatusCodes.Status500InternalServerError);
    }
}
```

Şekil 50. Fallback Policy-3

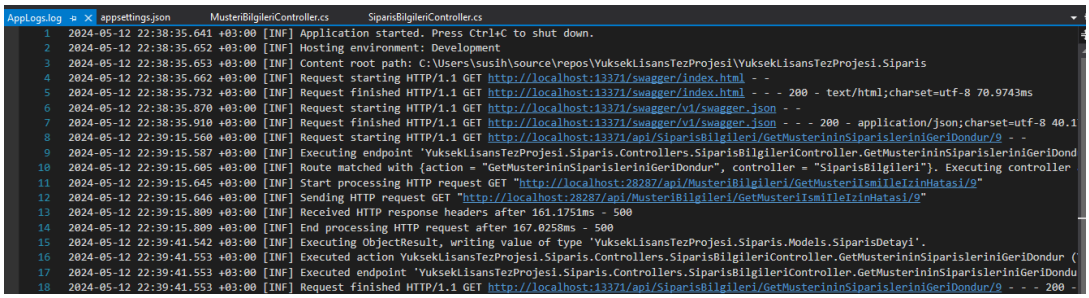
Kaynak: Yazar tarafından oluşturulmuştur.

HTTP durum kodları ile kontrolünü sağladığımız geri dönüş değeri alınan hatalar doğrultusunda arayüz üzerinden testi sağlanmıştır. Bu kapsamda, sipariş verilmesi sırasında müşteri ile yaşanan hataların iyileştirilmesi, kontrol altına alınması ve hızlı geri bildirimlerin sağlanması ile müşteri memnuniyetini artırır. Ayrıca sistem üzerinde yaşanan kararlılığın artışına bağlı olarak e-ticaret platformuna olan güven ile başarısına olumlu anlamda katkı sağlanmaktadır.



Şekil 51. Fallback Policy-4

Kaynak: Yazar tarafından oluşturulmuştur.



Şekil 52. Fallback Policy-5

Kaynak: Yazar tarafından oluşturulmuştur.

E-ticaret uygulamalarında sipariş servisinde ürünlerin ait olduğu müşterinin bilgilerinin alınması sırasında bir hata meydana gelebilmektedir. Meydana gelen bu hata kullanıcılar tarafından tekrar tekrar hatanın giderilmesi için istekte bulunabilmektedir. Başarısız olan isteğe ait, tekrar sayısı belirlenen değerleri aşması ya da tekrar tekrar yapılan denemeler sonucunda, sistemin kilitlenmesi durumunda önceden belirlenmiş bir süre boyunca istek gönderilmesini engellemek amacıyla Circuit Breaker politikası uygulamada kullanılmaktadır.

```
[HttpGet]
[Route("GetMusterininSiparislerindeDevreKesici/{musteriNumarasi}")]
0 references
public SiparisDetayi GetMusterininSiparislerindeDevreKesici(int musteriNumarasi)
{
    try
    {
        _httpClient = _httpClientFactory.CreateClient();
        _httpClient.BaseAddress = new Uri(url);
        var apiuri = "/api/MusteriBilgileri/GetMusteriIsmiIleIzinHatasi/" + musteriNumarasi;
        var sonuc = _circuitBreakerPolicy.Execute(() => _httpClient.GetStringAsync(apiuri).Result);
        _siparisDetayi.MusteriIsmi = sonuc;
        return _siparisDetayi;
    }
    catch (Exception ex)
    {
        _logger.LogError(ex, "Sefer Gerçekleşti");
        _siparisDetayi.MusteriIsmi = "Müşteri adına şu an için ulaşılamadı";
        return _siparisDetayi;
    }

}
```

Şekil 53. Circuit Breaker Policy-1

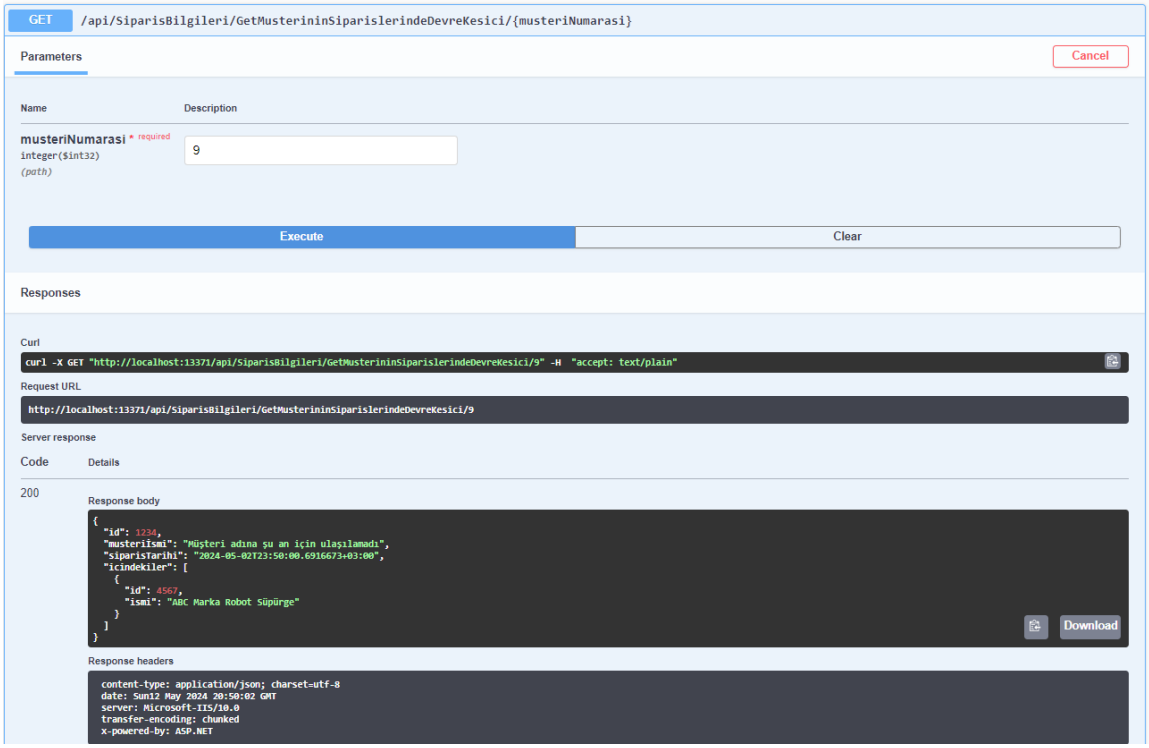
Kaynak: Yazar tarafından oluşturulmuştur.

Uygulama üzerinde tasarlanan sisteme göre müşteri servisinden 3 defa hata dönmesi durumunda, 1 dakikalık süreyle servise istek gönderilmemesi sağlanmaktadır. Bu istek gönderilememe durumunu uygulama üzerinde kullanıcı dostu arayüzler ile hata mesajlarının gösterilmesi sağlanabilmektedir.

```
private static CircuitBreakerPolicy _circuitBreakerPolicy;
0 references
public SiparisBilgileriController(ILogger<SiparisBilgileriController> logger, IHttpClientFactory httpClientFactory)
{
    _logger = logger;
    _httpClientFactory = httpClientFactory;

    if (_siparisDetayi == null)...

    if (_circuitBreakerPolicy == null)
    {
        _circuitBreakerPolicy = Policy.Handle<Exception>().CircuitBreaker(3, TimeSpan.FromMinutes(1));
    }
}
```

Şekil 54. Circuit Breaker Policy-2

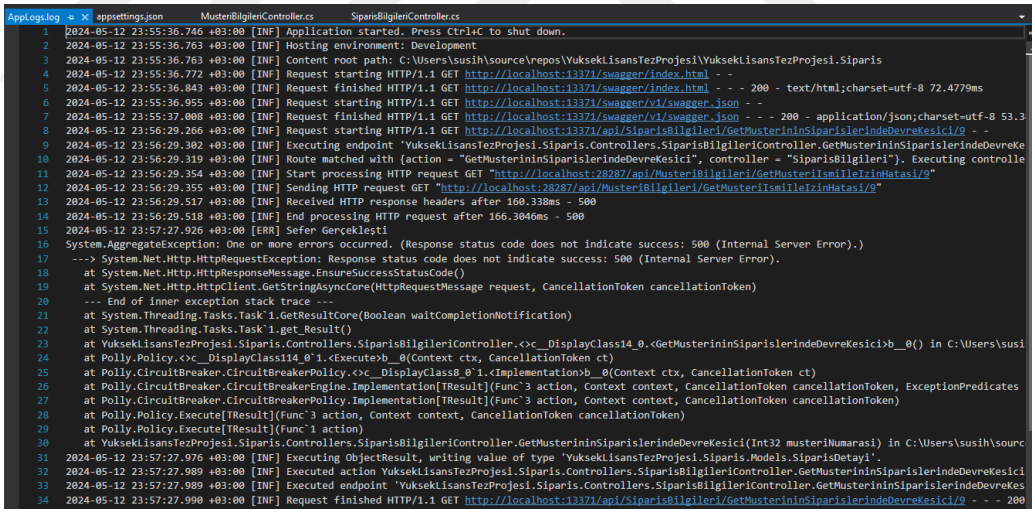
Kaynak: Yazar tarafından oluşturulmuştur.



Şekil 55. Circuit Breaker Policy-3

Kaynak: Yazar tarafından oluşturulmuştur.

Ardışık olarak yaşanan hataların düzeltilmesi zaman alabilmektedir. Bu sebeple, uygulama üzerinde API'lerin çalıştırılmasının ardından açılan arayüz **"/api/SiparisBilgileri/GetMusterininSiparislerindeDevreKesici/{musteriNumarasi}"** metodunda müşterinin numarasının girilmesinin ardından meydana gelen hata applog. dosyası üzerinde aşağıda gösterilmektedir. Swagger arayüzünde görüntülenen bu dönüş değeri app.log içerisinde deneme yapıldığı ve ardından geri dönüş değerinin 500 durum kodu alındıktan sonra bu politikanın uygulanması ile 200 geri dönüş değeri elde edilmiştir. Çünkü, serviste yaşanılan sorun mevcut iken geçen sürenin ardından sorun çözüme kavuşmuş ve kullanıma hazır hale gelmiştir. Bu kapsamda, sürekli hatalar ile karşılaşan bir sistemin belirli bir süre ile yanıt beklenmesinin kapatılmasıyla sistemin kullanıcılar tarafından zaman ve kaynak tasarrufu yapmasını sağlamaktadır. Kullanıcılara hata mesajlarının açıklayıcı bir şekilde gösterilmesiyle e-ticaret platformlarında iletişim aracı olarak kullanılan müşteri hizmetlerine olan talebi azaltır.



Şekil 56. Circuit Breaker Policy-4

Kaynak: Yazar tarafından oluşturulmuştur.

SONUÇ

Bu çalışmada, öncelikle monolitik ve mikroservis mimarisinin özelliklerinden, avantaj ve dezavantajlarından detaylı bir şekilde bahsedilmiştir. E-ticaret uygulamalarında, mikroservis mimari karmaşık yapıları belirli işlevlere göre ayırabilmesi ve ölçeklendirmesi sebebiyle sıklıkla tercih edilmektedir. Mikroservis mimari, birbirinden bağımsız çalışan servisler aracılığıyla meydana gelen hata durumunda başka bir servisin çalışmasını engellememektedir.

E-ticaret uygulamalarında siparişi veren müşteriye ait bilgilere ulaşılması sırasında meydana gelen hataların önüne geçilebilmesi için .NET Polly kütüphanesinin entegrasyonu ile uygulama geliştirildi. Bu uygulama ile mikroservis mimarisinin e-ticaret uygulaması üzerinde kullanılması sırasında meydana gelebilecek örnek senaryolar ele alındı. Bu senaryolar kapsamında, kullanıcıların tatmin edici şekilde bir alışveriş deneyimi yaşaması için hata mekanizmalarının uygulanabilirliği sağlandı.

Hata yönetim kütüphanesinin dört farklı politikası uygulandı. Bu kapsamda, uygulama üzerinde yaşanan hatanın giderilmesine dair yeniden denenebilirliği, geri dönüş değeri ile yedek bir değer gösteriminin sağlanmasını, hata sırasında bir süre sisteme erişimin kesilebilirliği, servisten yanıt alınamaması durumunda belirlenen süre sonunda isteğin sonlandırılması ile kullanıcılara sorunsuz bir deneyim yaşatmak için geliştirmeler sağlanmıştır. Kütüphanenin kullanımı ile sistem karmaşıklığı, veri tutarlılığı ve performans kayıpları gibi önemli konuların e-ticaret uygulamalarında etkin bir şekilde yönetilmesi sağlanmıştır.

Sonuç olarak, geliştirilen uygulama ile e-ticaret platformlarında sipariş sürecinde müşteri verilerine dair oluşan hataların önlenebilmesi için çözümler sunulmaktadır. Bu çözümler ayrıca geniş kapsamlı olarak e-ticaret uygulamalarında başka servislerde de uygulanması durumunda çözüme katkı sağlayabilmektedir. Müşteri memnuniyetinin artışının sağlanması ile birlikte e-ticaret platformlarına olan güveni de günden günden arttıracaktır.

Gelecekteki çalışmalarda, müşterilerin deneyiminin optimize edilerek yapay zeka, veri analitiği ve makine öğrenmesi yöntemleri ile e-ticaret uygulamalarında daha kişiselleştirilmiş deneyimlere odaklanılabilir. Sanal gerçeklik, artırılmış gerçeklik gibi yeni nesil teknolojiler ile birlikte siparişlerin oluşturulması sırasında yaşanan hata durumlarında platform üzerinden alışveriş yapılması yerine mağaza içinde uzaktan gezinti yaparak sipariş oluşturulması sürecine odaklanılabilir. Müşterilere daha esnek ve kontrollü alışveriş deneyimi yaşatılabilir. Bu şekilde, müşteri deneyiminin iyileştirilmesi ve yenilikçi yaklaşımların yaygınlaşması beklenmektedir.

KAYNAKÇA

Aksu, E. B., Gündüz, C. ve Ayanoğlu, E. (2013). Farklı mobil platformlar üzerindeki servis tabanlı mimari (soa) yaklaşımı: Elektronik uçuş çantası vaka çalışması. *Akademik Bilişim Konferansı Bildileri, Antalya.*

Altınkaya, C. (2022). *Üniversite bilgi sistemleri için REST tabanlı bir web servis platformunun tasarımı ve geliştirilmesi* [Yüksek lisans tezi]. Atatürk Üniversitesi.

Asrowardi, I., Putra, S, D. ve Subyantoro, E. (2020). *Designing microservice architectures for scalability and reliability in e-commerce.* (s. (Vol. 1450, No. 1, p. 012077). IOP Publishing). Journal of Physics: Conference Series.

Cebeci, K. (2019). Design of a queue-based microservices architecture and performance comparison with monolith architecture [Doktora Tezi]. Marmara Universitesi.

Darmawan, B., Richter, C., Lima, G., Salazar, M. E., Poppe, M. ve Veetil, V. E. (2008). Power systems and soa synergy. *IBM Redbooks publication.*

Dervişi, F. (2022). *Açık bankacılık sistemlerinde monolitik mimariden mikroservis mimariye geçiş* [Yüksek lisans tezi]. Trakya Üniversitesi.

Dmitry, N., Manfred, S. S. (2014). *On micro-services architecture* (s. 2(9), 24-27). International Journal of Open Information Technologies.

Dragoni, N., Giallorenzo, S., Lafuente, A. L., Mazzara, M., Montesi, F., Mustafin, R. ve Safina, L. (2017). Microservices: yesterday, today, and tomorrow. *In book: Present And Ulterior Software Engineering.*

Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P. ve Berners-Lee, T. (1999). *Hypertext transfer protocol--HTTP/1.1.* (No. rfc2616).

Galster, M. ve Angelov, S. (2016). What makes teaching software architecture difficult? (s. pp. 356-359). *2016 IEEE/ACM 38th International Conference on Software Engineering Companion.*

George, C., Jean, D., Tim, K. ve Gordon, B. (2012). D*istributed systems*. Concepts and Design 5th Ed.

gRPC Authors. (2024). *Introduction to gRPC.* 5 Şubat 2024 tarihinde https://grpc.io/docs/what-is grpc/introduction/ adresinden edinilmiştir.

Goldberg, A., Buff, R., Schmitt, A. (1998). A comparison of HTTP and HTTPS performance. Computer Measurement Group, CMG98, 8.

Gos, K. ve Zabierowski, W. (2020). The comparison of microservice and monolithic architecture. *2020 IEEE XVIth International Conference on the Perspective Technologies and Methods in MEMS Design (MEMSTECH).*

Gördesli, M. ve Varol, A. (2022). Comparing interservice communications of microservices for e-commerce industry. (s. pp. 1-4). *2022 10th International Symposium on Digital Forensics and Security (ISDFS) IEEE.*

Hasselbring, W. ve Steinacker, G. (2017). Microservice architectures for scalability, agility and reliability in e-commerce (s. pp. 243-246). *2017 IEEE International Conference on Software Architecture Workshops (ICSAW).*

Ko, R, K., Kirchberg, M, Lee, B.S ve Chew, E. (2012). Overcoming large data transfer bottlenecks in restful service orchestrations. *2012 IEEE 19th International Conference on Web Services.*

Kocaman, Y. (2018). M*ikroservis tabanlı ödeme sistemi tasarımı ve gerçekleştirilmesi* [Yüksek lisans tezi]. Yeditepe Üniversitesi.

Konigsburg, E., Pant, R., Kvochko, E. (2014). *Security Challenges in an Increasingly Tangled Web.* 10 Şubat 2024 tarihinde https://open.nytimes.com/ adresinden edinilmiştir.

Kyryk, M., Tymchenko, O., Pleskanka, N. ve Pleskanka, M. (2022). *Methods and process of service migration from monolithic architecture to microservices.* (s. pp. 553-558). 2022 IEEE 16th International Conference on Advanced Trends in Radioelectronics, Telecommunications and Computer Engineering (TCSET).

Lewis, J. ve Fowler, M. (2014). *Microservices: A definition of this new architectural term.* 21 Ocak 2024 tarihinde https://martinfowler.com/articles/microservices.html adresinden edinilmiştir.

Microsoft. (2024). *Application resiliency patterns.* 10 Şubat 2024 tarihinde https://learn.microsoft.com/en-us/dotnet/architecture/cloud-native/application-resiliency-patterns adresinden edinilmiştir.

Microsoft. (2024). *Microservices architecture style.* 24 Şubat 2024 tarihinde https://learn.microsoft.com/en-us/azure/architecture/microservices/design/ adresinden edinilmiştir.

Natran. (2023). *Evolution of HTTP Over the years.* 17 Ocak 2024 tarihinde https://developer.mozilla.org/ adresinden edinilmiştir.

Pandurang. (2022). *How does HTTP protocol work.* 20 Ocak 2024 tarihinde https://pandurangpatil.medium.com/how-does-http-protocol-work-426b1a4158f3 adresinden edinilmiştir.

Ponce, F., Márquez, G. ve Astudillo, H. (2019). *Migrating from monolithic architecture to microservices: A Rapid Review* (s. pp. 1-7). 2019 38th International Conference of the Chilean Computer Science Society (SCCC).

Ramanathan, R. ve Korte, T. (2014). *Software service architecture to access weather data using RESTful web services* (s. pp. 1-8). Fifth International Conference on Computing, Communications and Networking Technologies (ICCCNT).

Richardson. (2017). *Pattern: Microservice architecture.* 27 Şubat 2024 tarihinde http://microservices. io/patterns/microservices. html. adresinden edinilmiştir.

Richardson, L., Ruby, S. (2007). *RESTful Web services.* Sebastapol, CA: O'R; eilly Media.

Shafabakhsh, B., Lagerström, R., Hacks, S. (2020). Evaluating the impact of inter process communication in microservice architectures. QuASoQ@ APSEC.

Singleton, A. (2016). *The economics of microservices* (s. 3(5), 16-20). IEEE Cloud Computing.

Terra, R., Valente, M. T. ve Bigonha, R. S. (2012). *An approach for extracting modules from monolithic software architectures* (s. pp. 1-18). IX Workshop de Manutenção de Software Moderna (WMSWM).

Van Steen, M. ve Tanenbaum, A. (2017). Distributed systems. Leiden, The Netherlands: Maarten van Steen.

Velepucha, V. ve Flores, P. (2023). *A survey on microservices architecture: Principles, patterns and migration challenges.*

Wang, T., Zhang, W., Xu, J. ve Gu, Z. (2020). *Workflow-aware automatic fault diagnosis for microservice-based applications with statistics.* (s. 17(4), 2350-2363). IEEE Transactions on Network and Service Management.

Wang, X., Zhao, H., Zhu, J. (1993). *GRPC: A communication cooperation mechanism in distributed systems.* (s. 27(3), 75-86). ACM SIGOPS Operating Systems Review.

Wei, P., Hong, Z. ve Shi, M. (2016). Performance analysis of HTTP and FTP based on OPNET. 2016 IEEE/ACIS 15th International Conference on Computer and Information Science (ICIS).

Zhou, vd., 2018). F*ault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study.* IEEE Transactions on Software Engineering,47(2), 243-260.